USENIX Sixth Annual Tcl/Tk Conference Proceedings *San Diego, California* *September 1998*

conference

proceedings

The Sixth Annual Tcl/Tk Conference Proceedings

San Diego, California
September 14–18, 1998

Sponsored by
The USENIX Association

USENIX®

The Advanced Computing
Systems Association

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $22 for members and $28 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

ISBN 1-880446-98-7

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the Sixth Annual Tcl/Tk Conference

September 18-24, 1998
San Diego, California

Table of Contents

The Sixth Annual Tcl/Tk Workshop

September 14-18, 1998
San Diego, California

NBC's GEnesis Broadcast Automation System: From Prototype to Production

Stephen J. Angelovich
NBC Broadcast and Network Operations

Kevin B. Kenny
Brion D. Sarachan
GE Corporate Research & Development
kennykb@crd.ge.com

Abstract. GEnesis is a system in use at the NBC television network for automating the composition and distribution of video. It works in a mission critical environment; a system failure could potentially result in a substantial loss of revenue for the network. Tcl/Tk has been an integral part of the operator interface and data handling portions of the GEnesis system from the earliest stages of prototyping. We originally planned to replace the system prototype based on Tcl/Tk with a production system built in a compiled, object-oriented language and using commercial component software. After the prototype phase was completed, the developers and management together decided to keep numerous system components in Tcl, while migrating some complex and performance-critical functions from Tcl to a C++ message passing architecture. This paper discusses that decision and presents our experience with converting the prototype into a fully functional system.

1. Introduction

GEnesis is an upgrade to the NBC television network to support the requirement for digital video and to increase the network's capacity from ten simultaneous streams of video to forty. It is an ambitious control system; it includes roughly 400 computer-controlled devices for processing, storing, and routing video. Among its components are some sixty devices for video storage and processing that include RAID arrays totaling several tens of terabytes of disk space, satellite uplink/downlink controls at over two hundred stations, and high-bandwidth digital video routers to interconnect the devices. The system does not run production studios nor transmitters at the local stations, but handles all the tasks needed in between: video storage and playback, combining video segments into an integrated stream, adding special effects, doing voice-overs, and managing the satellite distribution system for over 200 NBC affiliate stations. An overview of the system appears in Figure 1.

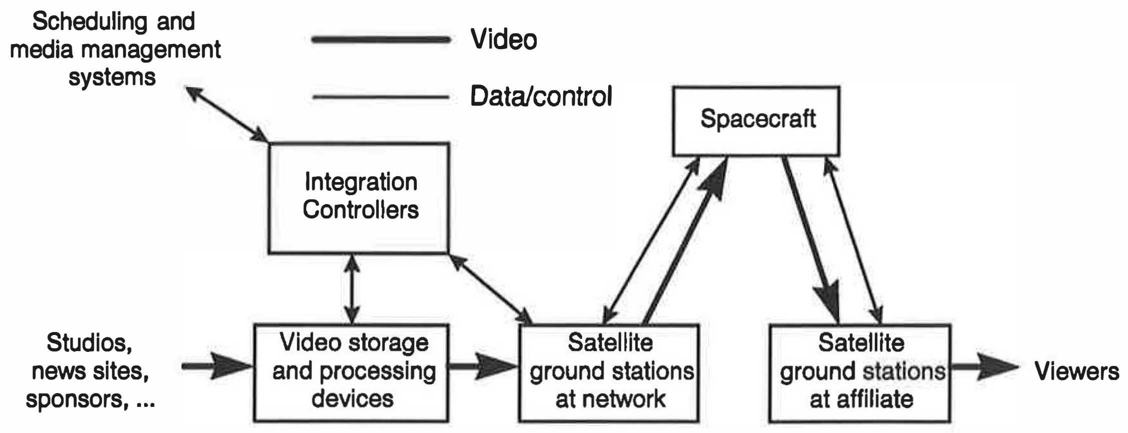

Scheduling and media management systems
Video
Data/control
Spacecraft
Integration Controllers
Studios, news sites, sponsors, ...
Video storage and processing devices
Satellite ground stations at network
Satellite ground stations at affiliate
Viewers

Figure 1. GEnesis system overview

Tcl/Tk was chosen over four years ago to build prototypes for several user interfaces in the GEnesis system. The idea at the time was to use Tcl/Tk to build a working model, test new ideas, and expose a variety of proposed user interfaces to the system operators. At the time, there were no plans to preserve any of the prototype into the deployed system.

As we described in the 1995 Tcl/Tk Workshop [SSZ95], Tcl was well suited forprototyping the GEnesis application. After the publication of our previous paper, prototype development continued to proceed rapidly. We completed 1995 with a comprehensive GEnesis prototype, including simulators for digital video servers and routers, all implemented using Tcl/Tk, with database access using `sybtcl` [Poin97] and interprocess communication using `Tcl-DP` [SRY93].

When we started building the production system, we decided to take the radical step of keeping Tcl/Tk when building the Integration Controllers — the computers that accept the network schedule from database systems upstream, provide the user interface to the network operators, and deliver commands to the video devices. Performance-sensitive components, and components that required access to third-party interfaces, were migrated to C++. The user interface continued to be built with Tcl/Tk. The C++ code supported extensive accessor functions that allowed it to be configured and commanded from Tcl. This design had several advantages. It allowed extensive configurability: the same executable is now used to build six different types of operator station, plus a wide array of plug-in test harnesses. It allowed us continued access to the Tk text and canvas widgets; no other user interface toolkit seems to have anything nearly as effective for customizable user interfaces. It allowed us to have a working version of the system at all times; there was no early period when the system was "still under development" with nothing available to demonstrate. Perhaps most important, Tcl's embeddable nature allowed us to develop interfaces to the actual devices even though they came from several vendors and had widely varied interface conventions.

2. From prototype to deployed system

We continued development with a natural iteration among prototype development, use case definition, requirements understanding, and architecture development. The development was much the same as the "star" approach described in [HH89]. We felt that this development process was ideal for a project like GEnesis, in which the resulting system was to be much different from the system being replaced; understanding and defining the requirements was part of the ongoing development process. The requirements could not possibly have been laid out at the outset according to the traditional "waterfall" development process. The "star" approach has allowed us to converge on the requirements in close collaboration with our colleagues at NBC.

One of the keys to our success in this project was to migrate gracefully from prototype to deployed system. We have had a working system at all times since we began in late 1994, with increased levels of functionality, reliability, and performance. We have never allowed ourselves the luxury of a "big bang" that would break the system for an extended period. In the later phases of development, we have constructed daily builds of the software, complete with automated installation scripts. Nearly every build has been tested immediately at NBC.

As stated above, our initial prototypes were implemented entirely using Tcl. This section summarizes some of the key steps as we migrated to a highly reliable, production quality software system.

2.1 Message passing architecture

The Integration Controller supports a complex set of functions, which include such things as simulating the device-level execution of the network television schedule, and reconciling the simulation against status returned by the actual devices, to determine whether execution was correct. We originally implemented much of this logic in Tcl. This exercise was highly valuable for prototyping the logic and understanding the requirements.

The complexity of this implementation in a scripting language did eventually become unwieldy. In particular, we fell victim to namespace pollution. Global arrays (representing, for instance, all events using a given device in time sequence, all events referring to a particular video clip, and so on) proliferated, and maintaining a dictionary of their names became unwieldy. Moreover, the arrival of each event resulted in the execution of many thousands of lines of Tcl code to maintain the data structures. The pure Tcl implementation had trouble keeping up with its workload.

We developed a C++ object-oriented architecture, known as "NetSys," to provide a maintainable and extensible infrastructure for the Integration Controller's processing. (NetSys itself has many interesting aspects to its design, which are beyond the scope of this paper.)

2.2 Extensible Tcl event loop

The Integration Controller is an event-driven application that receives data, control signals, and device status through a variety of communication protocols, including TCP and UDP sockets and proprietary protocols. We were able to make appropriate extensions to and callbacks from the Tcl event loop to support all of these interfaces, in addition to the user interface events that it already supported.

Integrating additional event sources into the Tcl event loop proved to be extremely simple. Only two areas were really cause for concern. The first of these was the fact that non-blocking interfaces are unnatural on the Windows platform, where multithreaded applications are the rule rather than the exception. Some of the third-party interfaces that we must use, therefore, freeze the event loop for longer than we would like while they are performing their tasks. We deal with this situation by the expedient of putting the code that uses these interfaces into separate processes, and communicating with them using sockets. If the event loop stops responding temporarily, our message queues hold the traffic for these processes until it starts again. We may revisit this decision when the thread-safe Tcl core becomes available in release 8.1.

The second cause for concern is the stability of the interfaces. It seems that any C code that uses the Tcl event management primitives breaks with every release of Tcl, since the protocols change so rapidly. Tracking these changes over the life of the project was a major headache, and some of the changes seemed gratuitous. We hope that the multi-threaded notifier will represent the last round of major incompatible changes.

2.3 Uniform interfaces

On several occasions we replaced prototype functionality while keeping the interfaces constant. This greatly facilitated our goal of always having a working system. For example, the early prototype code often used Tcl associative arrays as data structures. When moving to C++, we have often provided bi-directional mappings from C++ objects to Tcl associative arrays, allowing for interoperability between the Tcl and C++ implementations.

The NetSys library provides a uniform messaging interface to socket connections, database access, device drivers, user interface displays, and internal processing. By using Tcl interfaces like the ones presented here, functionality can be organized in different configurations through Tcl scripts which configure the NetSys message handlers.

The mapping between objects and associative arrays turns out to be surprisingly easy to do. Each C++ class has a few stylized static methods that hook it into Tcl. The `constructFromTclArray` method (Figure 2) extracts the values from the Tcl array and invokes the C++ constructor.* It then installs the newly-created object into the Tcl command namespace.

The `tclCommandDeleteFunction` method (Figure 3) is a trivial connection to the destructor.

The `tclCommandFunction` method provides whatever method calls are needed from Tcl. One interesting technique that we use frequently is to pass a C++ object by reference to a method in another C++ object. The Tcl interface is to pass the name of the Tcl command that represents the object. The `Tcl_GetCommandInfo` library function is used to validate that the string is the name of an object of the correct type, by examining the command function pointer (Figure 4).

When filling an associative array with the contents of a C++ object, we have usually found it more convenient to create a list of alternating keywords and values, return that list to Tcl, and use the `[array set]` command to install the values in the Tcl array. This trick meant that the C++ code did not need to be prepared to deal with possible error status returned by `Tcl_SetVar2`.

2.4 User interface evolution

Several of our prototype user interface screens were shown in our earlier paper [SSZ95]. One result of our "star" development process was to continually refine these screens based on user feedback. Tk provided an ideal tool for easy changes to the user interface. We wholeheartedly agree with the sentiments expressed by Brian Kernighan in [KERN95]; the Tk text and canvas widgets provide power and flexibility at least as good as anything else on the market.

One specific change that the NBC users requested early was to replace our early "busy" screens with a simpler layout, and provide a rich assortment of application views as "notebook tabs," as is familiar in many modern Windows-based applications. It might not have been inordinately difficult to implement this look and feel using the canvas widget (as Harrison and McLennan do in [HM98]), but it would have been time-consuming. Instead, we took advantage of the Tcl community and adopted Ioi Kim Lam's Tix widget set, which provided us with a notebook widget off-the-shelf.

* All of the illustrative code is targeted to Tcl 7.6, which is now obsolete. The GEnesis project continues to use it because it works adequately well, and the benefits to be gained from moving forward to the Tcl 8 object APIs are not yet worth the effort of converting the C++ code.

```
int
myClass::constructFromTclArray (Tcl_Interp* interp,
                                char* arrayName)
{
    char* string;          // working storage
    int parameter1;        // constructor parameters
    char* parameter2;
    if (string = Tcl_GetVar2 (interp, arrayName,
                              "parameter1", TCL_LEAVE_ERR_MSG) == NULL)
      return TCL_ERROR;
    if (Tcl_GetInt (interp, string, &parameter1)) != TCL_OK)
      return TCL_ERROR;
    if (parameter2 = Tcl_GetVar2 (interp, arrayName, "parameter2",
                                  TCL_LEAVE_ERR_MSG) == NULL)
      return TCL_ERROR;
    myClass* newObject = myClass (parameter1, parameter2);
    char instName[24];
    sprintf (instName, "myClass_%p", (void*) &newObject);
    Tcl_CreateCommand (interp,  instName, myClass::tclCommandFunction,
                       (ClientData) newObject,
                        myClass::tclCommandDeleteFunction);
    Tcl_SetResult (interp, instName, TCL_VOLATILE);
    return TCL_OK;
}
```

Figure 2. Constructing a C++ object from a Tcl array

```
static void
myClass::tclCommandDeleteFunction (ClientData clientData)
{
    delete (myClass*) clientData;
}
```

Figure 3. Destroying a C++ object from Tcl

```
Tcl_CommandInfo info;
if (Tcl_GetCommandInfo (interp, commandName, &info) == 0) {
    Tcl_AppendResult (interp, name, ": no such command", (char*) NULL);
    return TCL_ERROR;
  }
  if (info.proc != &DesiredClass::TclCommandFunction) {
    Tcl_AppendResult (interp, name,
                      " is not an instance of DesiredClass",
                      (char*) NULL);
    return NULL;
  }
  return (NodeBaseClass*) info.clientData;
```

Figure 4. Type-checking an object passed by name from Tcl

2.5 Port from Solaris to Windows NT

Another major benefit we derived from Tcl/Tk was portability. Two years into the project, NBC decided to change computer platforms from Solaris to Windows NT. The porting effort was minor, thanks to the fact that Sun's first Windows port was released just in time. We have evolved a team development environment that uses Windows-based tools (Microsoft Developer's Studio, MKS Source Integrity, Purify, pcAnywhere, and other tools) and suits our purposes well. Had we not chosen Tcl/Tk early on (the original proposal called for implementation with OpenWindows and Motif), the unexpected port to Windows could easily have been a showstopper.

3. Using Tcl in critical systems

The GEnesis system had tight requirements in a number of critical areas. It required very high *reliability:* system failures must be few or nonexistent (the eventual system is targeted to have less than 20 minutes of unscheduled outage in a year's operation). It has several tight *performance* requirements: it has a complex graphical user interface that may have several dozen objects updated in a second. It also has a *longevity* requirement: the operator's workstation cannot be interrupted more than about once a week, and the applications have to be able to run that long without restarting. After a small number of issues were resolved, as summarized below, Tcl/Tk together with custom extensions provided a robust platform which supported the reliability, performance, and longevity requirements of GEnesis.

3.1 Reliability

Our experience has been that Tcl/Tk has presented few reliability problems. In the course of deploying the system, we discovered several bugs in the Tcl/Tk implementation that resulted in program crashes. In all cases, the Sun staff were able to find and correct the problems in short order. In the last several months of operation (since the bugs that we encountered were fixed or worked around), no system failures have occurred that can be ascribed to problems with the Tcl/Tk core.

3.2 Performance

As we have already discussed, the Tcl interpreter was too slow for various operations in the system that involved complex data structures. We reworked these operations in C++. Ultimately, the parts of the system dealing with device control and data management evolved to being built entirely from C++ objects, with Tcl used as a configuration language to string these objects together.

Performance of the graphical user interface has been a more difficult problem; we ran into several entirely unexpected performance problems, some of which proved hard to characterize. The first of these is simply the vast amount of memory allocation activity that Tcl requires (another culprit here is the Rogue Wave libraries [RW96], which we use extensively). Our initial development environment used the "debugging" versions of `malloc` and `free`, which cleared memory and had basic integrity checking. When we replaced these with the non-debugging versions and instrumented the code, we saw a 50% performance gain for the common operations of starting and stopping video clips. We have not resorted to the non-debugging libraries in practice, feeling that the additional checking gives us a safety net. Nevertheless, the temptation is there, and we may succumb to it at some time in the future.

A second user-interface performance issue is the Tk console on Windows NT, which is simply too slow to use for more than a tiny volume of output. In addition, its performance appears to degrade rapidly as more text is added to it — a console with a few hundred lines of text in it is noticeably sluggish. Using the console as a message log is a longevity issue as well, since printing to the console consumes memory rapidly, using the better part of a kilobyte of heap space to display a one-line message. One part of our testing procedure is to inspect the Tcl code rigorously to make sure that all `puts` directives to the console are removed. In addition, to cover any console output that may escape this net, we have added to our initialization a script that looks like Figure 5.

```
proc keepConsoleClean {} {
  console eval {
    .console delete 1.0 end-1001
  }
  after 15000 keepConsoleClean
}
keepConsoleClean
```

Figure 5. Script to limit the console display

Another performance issue that recurred several times in the course of development was the management of tags in the text widget. One central part of the GEnesis system is the display of lists: lists of video to play, list of files to transfer among the playback devices, lists of available video clips, and so on. These lists are displayed in columns in the text widget, and the operator is provided with a quick-and-dirty query mechanism in which a double-click in any item highlights all lines having the same value in the item's column. This mechanism allows very quick answers to questions like, "what events use playback device CDX-01?" or "at what times are we scheduled to run the latest *Third Rock* promo?"

Our original design for the system used multiple text tags on each line, one for each field value, and did the highlighting by changing the display attributes of those tags. When we tried this scheme with hundreds or thousands of lines, however, we found that it was totally unworkable. The double click sometimes required many seconds to produce results, and the entire user interface was frozen for that length of time. To avoid this issue, we developed a complex scheme of tag management, which is of sufficient interest that we present it separately in Section 4.

3.3 Longevity

Achieving the longevity needed for a system in 24×7 operation was also a challenge. The problem here was a variety of resource leaks. The initial prototype for the system [SSZ95], which was built in Tk 3.6 on a Unix system, was awful in this regard, because of the way that Xlib leaked resource identifiers, and often crashed within a few hours. We are grateful to John Ousterhout for having fixed this problem in release 4.0.) The problem of memory leaks on the X11 platform is still intractable (we have found that X11 servers from several vendors need to be restarted every week or so); fortunately, it appears to be less of an issue on Windows NT, where we routinely run Integration Controllers for weeks at a time.

Of course, we continually have to scrub our C++ code for leaks. One recurring issue with Tk results in memory leaks: the `Tk_Uid` data type. This data type is used internally to Tk for making single copies of the names of objects: widget names, text and canvas tag names, and so on. The copies are kept in a hash table, and pointers to the strings may be compared with a simple comparison of pointers rather than a full string comparison.

Unfortunately, the `Tk_Uid` scheme assumes that there will be some small fixed set of names. If any tags or widgets are assigned names based on their content or based on an incrementing sequence number, the corresponding `Tk_Uid` objects are created and never recycled, resulting in a constant memory drain. This problem hit us unexpectedly several times. The most difficult was in the area of text and canvas tag management. We wished to tag lines of text and canvas items so that we could rapidly find the items displaying particular data. Since some of the data are never reused (in particular, the start time of video events continuously advances), any obvious tag scheme caused `Tk_Uid` objects to be created endlessly. We eventually developed a complicated tag management scheme to deal with this problem.

We found ourselves unable to resolve some of the issues of managing `Tk_Uid` objects, and dealt with them by negotiating away requirements. In particular, the customer once requested that the title bar of the main user interface window show a summary status line, with the current time and various parameters relating to the load on the system. We spent half an hour or so coding up this functionality in Tcl, only to discover that the `wm title` command creates a `Tk_Uid` object for the window title. This leak consumed memory at a rapid enough rate to crash the application after only a few hours. When we analyzed the problem, we decided that the functionality was not worth the amount of time that it would take to fix the problem in the Tcl core. (To the best of our knowledge, this is the only Tcl/Tk bug that we encountered but didn't actually fix.)

4. Tk tag management

In order to address the problems of text and canvas tag management, we were forced to develop our own tag maintenance system that layered on top of Tk's system. The system was designed to meet the following constraints:

- Inserting and deleting tagged items must be fast.
- Every item must have a unique tag, so that no tag will span a large fraction of the text widget or canvas display list.
- Tags must be drawn from a limited set of identifiers that are recycled to avoid leaking `Tk_Uid` objects.

The solution that we chose was to represent the tag structure in a family of global variables. Each of these has a window path name as part of the array name; they are brought into scope with the `upvar` command:

```
upvar #0 next_unused_tag$w \
               next_unused_tag
upvar #0 free_tags$w free_tags
upvar #0 id_for_tag$w id_for_tag
upvar #0 tag_for_id$w tag_for_id
# ... and so on ...
```

When inserting an item into a widget, the first thing that the code must do is locate an available tag to label the item. It does so with code like the following:

```
set searchKey \
  [array startsearch free_tags]
if {[array anymore \
        free_tags $searchKey]} {
```

```
  set tag [array nextelement \
             free_tags $searchKey]
} else {
  set tag [incr next_unused_tag]
}
array donesearch \
        free_tags $searchKey
```

Similarly, when deleting an item, its unique tag must be recycled:

```
set free_tags($tag) {}
```

Note that the set of recycled tags is maintained as the subscripts of a Tcl array (whose values are immaterial), and not as a Tcl list. We do this to avoid yet another performance problem. If the system has been under heavy load and subsequently is unloaded, there may be a great number of recycled tags. The code that removes one tag from the list:

```
set freeTagList \
  [lrange $freeTagList 1 end]
```

would have to copy the entire list. Recycling N tags would take $O(N^2)$ CPU time. This behavior surprised the original programmer, whose experience with Lisp suggested that there should be a constant-time operation analogous to Lisp's `cdr` function.

The `id_for_tag` array is used to map the unique tags back to object names in our C++ system, and the `tag_for_id` array maps the object names to the tags. These arrays are updated whenever items are created and destroyed.

Changing the appearance of an item or group of items, and responding to event bindings, requires executing `foreach` loops to locate the affected objects. Surprisingly, this scheme is much faster than the naïve scheme of tagging each item with its attributes, in addition to avoiding the problem of `Tk_Uid` leaks.

5. Troubleshooting resource management problems

While working on performance and longevity issues, the Genesis team used and developed a number of tools to track down specific CPU "hot spots" and memory management problems.

5.1 CPU performance measurements

It is virtually impossible to make headway with performance problems without profiling the code to find where it is spending its time. Tcl code, alas, confuses most profiling systems. When addressing performance issues, we tried profiling at the C/C++ level using tools like the profiler included with Microsoft Visual C++ and Pure Quantify. These tools were invaluable in isolating certain problems in our C++ code, but were virtually useless for telling where a Tcl program spends its time. (We already knew that it spends its time in `Tcl_Eval`, thank you!)

The profiler supplied with NeoSoft's Extended Tcl [LeDi89] was more effective, since it showed the time consumption based on the Tcl call tree. It showed only the time spent locally to a procedure, however, rather than the time spent in the procedure and the ones it calls. Given that Tcl programs are usually structured as many small procedures with complicated interrelationships, isolating performance issues from this output was also not easy.

Finally, in desperation, we implemented our own data reduction atop the NeoSoft profiler. Our system (Figure 6) uses the hierarchical list widget in Tix [LAM95] to display the call tree, and shows the real and CPU time spent in the procedure, and in the procedure plus descendants, next to the procedure's entry. Browsing through the output in this form makes the trouble spots obvious. (It would still be nice to tie them to source file and line number. We hope that the forthcoming TclPro product will finally make it possible.)

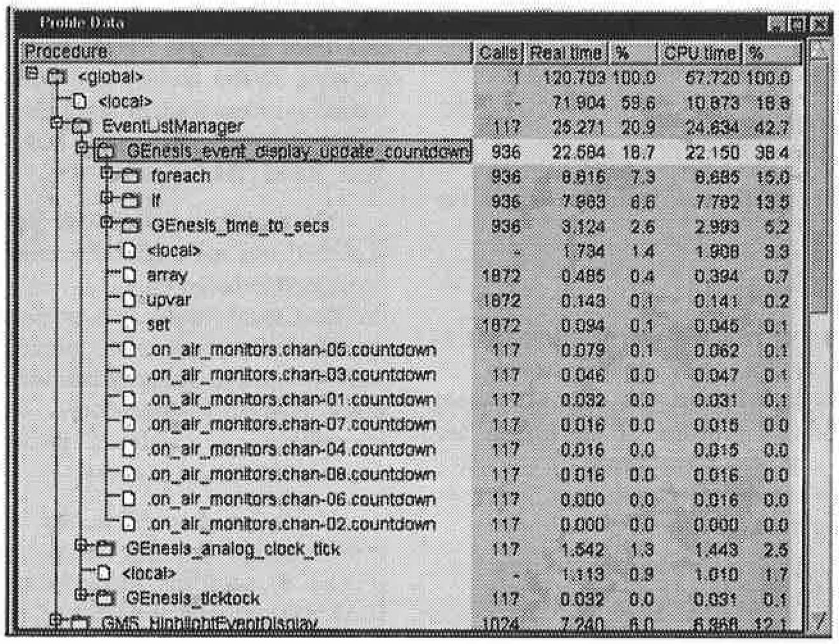
5.2 Resource leak identification

We use the Purify system extensively to track memory leaks. Unfortunately, we are handicapped by the extensive volume of the output that it produces. Not only Tcl, but also several other third-party software libraries that we use allocate memory at initialization time that is freed only by process exit rather than explicit calls to `free`. This memory is reported as "in use on exit" or "leaked on exit," exactly as if it were really leaked. We get around this problem by running the system for long periods with synthetic workloads, and then scanning the output of Purify for leaks involving large numbers of blocks.

In addition, we have developed a few little Tcl procedures that monitor the command and variable namespaces and report changes. These can be used to identify variables, array elements, and commands that are created and never deleted.

In our experience, the most difficult leaks involve the following areas:

- Traces established on nonexistent variables.
- Text and canvas tags that do not apply to any items in the widget.



Profile Data

Procedure	Calls	Real time	%	CPU time	%
<global>	1	120.703	100.0	67.720	100.0
<local>	-	71.904	59.6	10.873	18.8
EventListManager	117	25.271	20.9	24.634	42.7
GEnesis_event_display_update_countdown	936	22.584	18.7	22.150	38.4
foreach	936	8.816	7.3	8.685	15.0
if	936	7.983	6.6	7.782	13.5
GEnesis_time_to_secs	936	3.124	2.6	2.993	5.2
<local>	-	1.734	1.4	1.908	3.3
array	1872	0.485	0.4	0.394	0.7
upvar	1872	0.143	0.1	0.141	0.2
set	1872	0.094	0.1	0.045	0.1
.on_air_monitors.chan-05.countdown	117	0.079	0.1	0.062	0.1
.on_air_monitors.chan-03.countdown	117	0.046	0.0	0.047	0.1
.on_air_monitors.chan-01.countdown	117	0.032	0.0	0.031	0.1
.on_air_monitors.chan-07.countdown	117	0.016	0.0	0.016	0.0
.on_air_monitors.chan-04.countdown	117	0.016	0.0	0.015	0.0
.on_air_monitors.chan-08.countdown	117	0.016	0.0	0.016	0.0
.on_air_monitors.chan-06.countdown	117	0.000	0.0	0.016	0.0
.on_air_monitors.chan-02.countdown	117	0.000	0.0	0.000	0.0
GEnesis_analog_clock_tick	117	1.542	1.3	1.443	2.5
<local>	-	1.113	0.9	1.010	1.7
GEnesis_ticktock	117	0.032	0.0	0.031	0.1
GMS_HighlightEventDisplay	1024	7.240	6.0	6.968	12.1

Figure 6. Hierarchical browser for profile information

- Bindings established to nonexistent text and canvas tags, or to nonexistent binding tags.

The last two are easy to cause by accident, by deleting a canvas item or block of text while neglecting to delete its tags or bindings. These leaks are so difficult to locate because a product such as Purify finds only where they occur in the C source code, and cannot inform the programmer what Tcl code was being interpreted at the time. The Tcl system, moreover, does not export any interfaces that allow the programmer to write code to locate these abandoned items. We have occasionally resorted to the expedient of running the program for a while, stopping it under control of a debugger, and then using the debugger to examine Tk's internal data structures for these items. We do not recommend this activity as an amusing pastime.

6. Conclusions

Using Tcl/Tk has enabled us to develop and integrate systems rapidly. Its unparalleled embeddability has been a major productivity gain, particularly considering that the integration controller, as its name suggests, is intended to integrate the interfaces of video equipment from a number of different vendors.

A couple of worries that were raised during development were easy to address. The first has been a recurring concern that the interpretive nature of Tcl will make the performance of the system unacceptable. Since very little Tcl code is in the direct path of handling device commands and status, this objection really is not an issue. Moreover, the Tcl code in the user interface is a good bit faster than comparable interfaces implemented with toolkits other than Tk, even when the interpreter overheads are taken into account; Tk is *fast*. Another worry dealt with the availability of downstream support. This concern, too, was really not an issue. The source code for Tcl/Tk is freely available, and amounts to only a fraction of the total lines of code that the customer is already committed to maintaining in the GEnesis system. The extensive user community means that support will always be available somewhere.

Several technical issues, nevertheless, give us cause for long-term concern. One of them is stability of the application program interfaces. Each new release has broken our C++ code, and we have had to back and fill to get back on track. In practice, we have skipped every other release or more; we have used, in succession, releases 7.0, 7.4, and 7.6, and will most likely (as of this writing) bypass 8.0 in favor of 8.1. The more widely Tcl/Tk becomes deployed, the more consideration the Tcl/Tk development team will have to give to backward compatibility.

Another concern is the unexplained performance "hot spots". Some operations, such as copying canvases, are simply unusable. The Tcl console, too, becomes unworkably slow if thousands of lines are displayed. The text widget, in general, has to be tuned extremely carefully or it becomes too sluggish for our displays. Some of these behaviors could almost be characterized as "performance bugs" that could be corrected with some development effort, for instance, better indexing to support text tags that include large subsets of the widget.

In spite of these concerns, Tcl/Tk has been a wonderful framework for integrating systems. Each of our worries earlier in the program — for example portability to Windows NT, availability of appropriate database interfaces, and availability of native sockets — was available "just in time" for the next step in development, and we are confident that our remaining concerns will be addressed in the same way.

Acknowledgments

No project of this scale happens without the efforts of dozens of people. In particular, however, the authors would like to thank Steve Zahner, whose early advocacy at NBC was critical to getting Tcl/Tk accepted; David Rabinowitz, who has continued to support our development of the IC to production quality; John Ousterhout, Scott Stanton, Ray Johnson, Brent Welch, and others among the Tcl/Tk development team who freely provided hours of telephone support when we struggled with critical Tcl/Tk bugs; David Henderson, Chris Hammond, Mike Kinstrey, Dana Downey, Joe Amaral, Ralph Minerva, and the other GEnesis developers at GE and NBC.

References

[HH89] Hartson, H. R., and D. Hix. "Toward empirically-derived methodologies and tools for human-computer interface development." *Intl. J. Man-Machine Studies 31* (1989), pp. 477-494.

[HM98] Harrison, M., and M. McLennan. *Effective Tcl/Tk Programming*. Reading, Mass.: Addison-Wesley, 1998.

[KERN95] Kernighan, B.W. "Experience with Tcl/Tk for Scientific and Engineering Visualization." *Proc. Third Annual Tcl/Tk Workshop*, pp. 269–278. Berkeley, Calif.: USENIX, 1995.

[LeDi89] Lehenbauer, K., and M. Diekhans. "TclX - Extended Tcl: Extended command set for Tcl." Unpublished manual, available over the Internet at `http://www.neosoft.com/tclx/man/TclX.n.html`

[LAM95] Lam, I.K. "Designing Mega Widgets in the Tix Library." *Proc. Third Annual Tcl/Tk Workshop*, pp. 53–59. Berkeley, Calif.: USENIX, 1995.

[Poin97] Poindexter, T. "Sybtcl and Oratcl." In "Tcl/Tk Tools", M. Harrison, ed., *Tcl/Tk Tools*. Cambridge, Mass.: O'Reilly, 1997.

[RW96] *Tools.h++: Foundation Class Library for C++ Programming*. Part #RW-30-01-2-032596b. Corvallis, Ore.: Rogue Wave Software, 1996.

[SRY93] Smith, B.C., L.A. Rowe, and S.C. Yen. "Tcl Distributed Programming." *Proc First Tcl/Tk Workshop*, pp. 50-52. Berkeley, Calif.: University of California, 1993.

[SSZ95] Sarachan, B.D., A.J. Schmidt, and S.A. Zahner. "Prototyping NBC's GEnesis Broadcast Automation System in Tcl/Tk." *Proc. Third Annual Tcl/Tk Workshop*, pp. 251–260. Berkeley, Calif.: USENIX, 1995.

An Extensible Remote Graphical Interface for an ATM Network Simulator *

Michael D. Santos, P. M. Melliar-Smith, L. E. Moser
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
michael@alpha.ece.ucsb.edu, pmms@ece.ucsb.edu, moser@ece.ucsb.edu

Abstract

The University of California, Santa Barbara, is developing a 40 gigabit per second ATM switch as part of its Thunder and Lightning network project. To use the high bandwidth efficiently, new network protocols are being developed and simulated on the Thunder and Lightning network protocol simulator. Due to the extreme memory and computational requirements of the simulator, the display, unlike most Tcl/Tk interfaces, must be implemented as a distinct process capable of running on a remote machine. This paper discusses some of the issues that arise with such a physical separation of application and interface, and describes the implementation of the simulator's display application, with an emphasis on the use of the Tcl language. One module of the GUI for the simulator is discussed in detail, demonstrating the use of XDR (external data representation) with Tcl sockets to provide for cross-platform binary data exchange between the simulator and its display application. We also discuss our experience in building the simulator GUI and propose ways in which XDR might be incorporated into Tcl. We discuss some shortcomings of the canvas widget and describe mechanisms to overcome them.

1 Introduction

Advances in fiber-optic and VLSI technology have led to the emergence of very high-speed networks based on Asynchronous Transfer Mode (ATM) [1]. The Electrical and Computer Engineering Department of the University of California, Santa Barbara, in conjunction with Rockwell International Science Center, is currently building a 40 gigabit per second ATM switch as part of its Thunder and Lightning network project [2].

With the rapid increase in network bandwidth come new challenges for protocol development. Consider, for example, the situation in Figure 1 in which a user wishes to transmit data from San Diego, California to Boston, Massachusetts. In a typical ATM network, the user makes a request to reserve bandwidth from San Diego to Boston. The user's request traverses the network from San Diego to Boston, reserving bandwidth, and then returns to San Diego, informing the user of the capacity reserved, at which point the user can begin transmission. Due to the finite speed of light, this process takes approximately 40 milliseconds during which time 200 megabytes of data could have been transmitted into the Thunder and Lightning network if adequate capacity had been available.

Similarly, an ATM cell which is lost due to buffer overflow imposes a minimum delay of 40 ms before the retransmitted cell can be received at the destination. During this time, the sender could have transmitted another 200 megabytes of data into the network. To insert the retransmitted cell into the data stream properly (ATM guarantees in-order delivery of cells), the receiver must buffer this 200 megabytes of data while it waits for retransmission of the one lost cell. The sender faces similar buffering requirements as it must be able to retransmit a cell it sent at least 40 ms in the past (or 200 megabytes prior in the data stream). This is not a problem if the data source is a stable storage device, but may become a problem if, for example, the source of the data is a real-time measuring instrument.

As part of the Thunder and Lightning project, we are developing protocols [7, 8, 13, 14] that allow a sender to begin transmission without the lengthy reservation delay and yet provide lossless transmission. A network simulator [6] developed specifically

*This work was supported by DARPA Contract No. DABT63-93-C-0039, a National Science Foundation Graduate Research Fellowship, and a University of California Dissertation Year Fellowship.

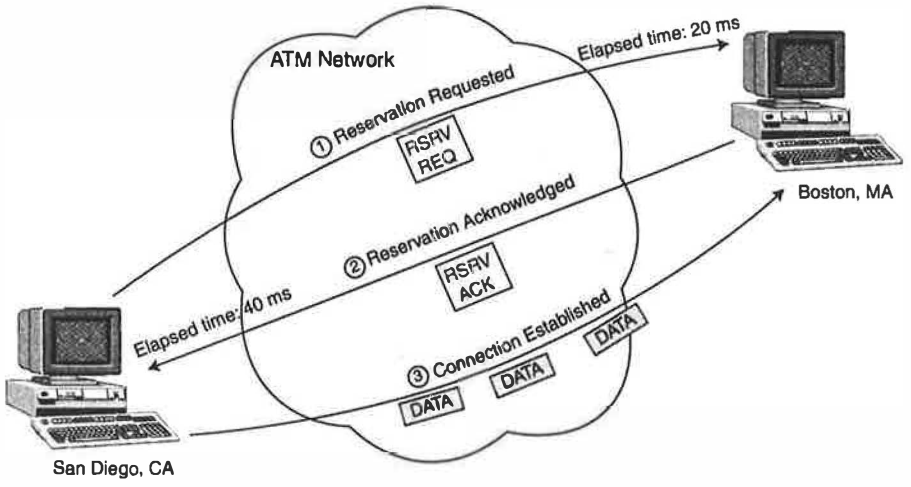


Figure 1: Connection establishment in a typical ATM network.

for Thunder and Lightning provides a testbed for the rapid prototyping, development, debugging and demonstration of these protocols.

To aid the protocol developer further, we have developed a graphical interface application which provides a view into the simulated environment. The display allows the user to manipulate the simulation in progress by inserting new events and altering existing events. Detailed protocol state information is also made available for debugging the protocol. This paper describes the design and development of this display program, emphasizing the use of Tcl/Tk. In addition to describing how Tcl/Tk has been an invaluable tool in rapidly developing this interface, we outline the mechanisms we developed to overcome the absence of certain features in the language.

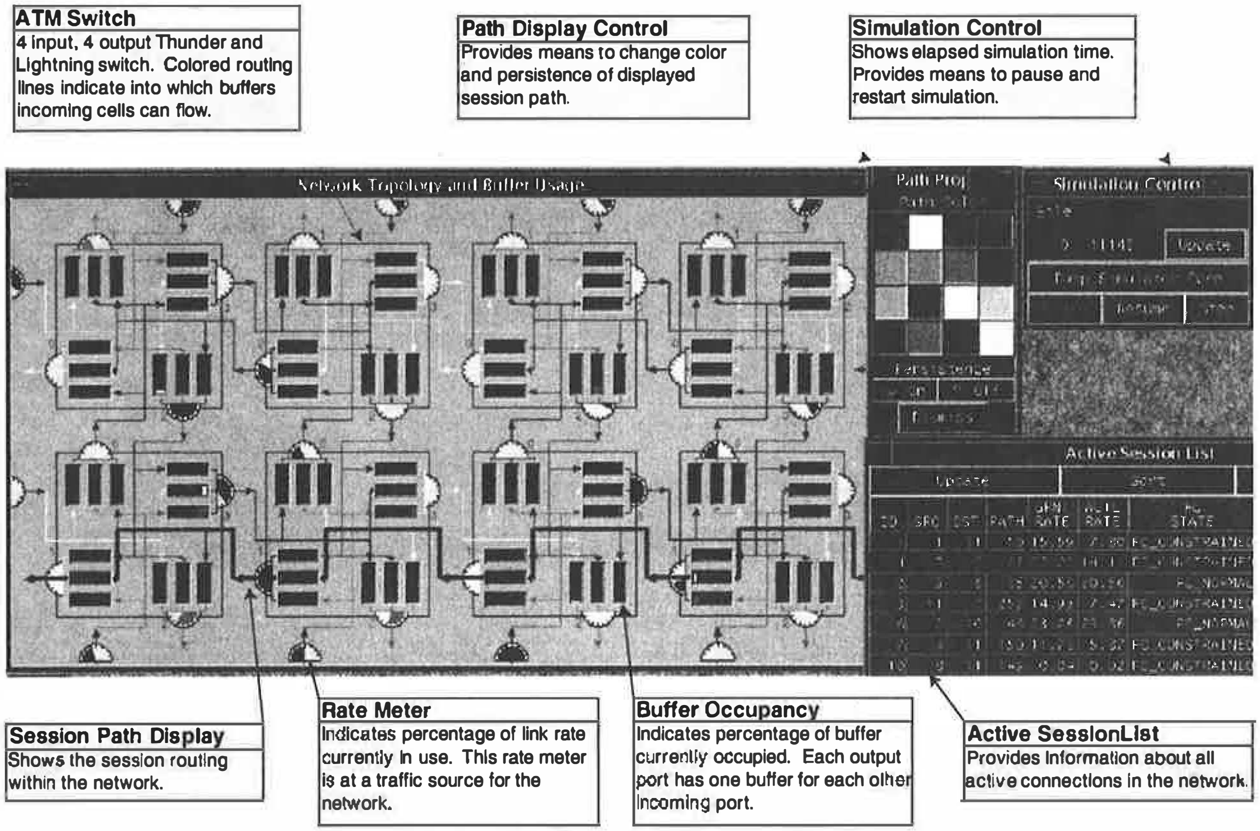
2 Design Issues for the Display Application

The Thunder and Lightning protocol simulator has stringent memory requirements as it must track every ATM cell in the simulated network to ensure that no cells are lost. When simulating a relatively simple 4×2 mesh of ATM switches and the data sources connected to it, the network simulator must keep track of over 2 million ATM cells, or 100 MB of data. While novel data representation techniques [6] allow us to reduce the memory requirements of the simulator significantly, it is clear that, while simulating more complex networks, the simulator may consume all of the available memory in its host machine. In addition, the simulation process is entirely compute-bound. Thus, any extra computation the host machine must perform negatively impacts the speed of the simulation.

Therefore, in designing the simulator display, we had the following goals:

- The display should reduce the memory available to the simulator as little as possible,
- The computational impact on the simulator should be negligible when the display is not in use,
- The display should be as flexible as possible so as to allow features to be added or modified as the protocol specification is changed, and
- The display should be reusable as a monitor for the real Thunder and Lightning switch.

We minimized the memory requirement of display code within the simulator by making the user interface a separate application. Whereas a typical Tcl/Tk application integrates the user interface into the application, we split the application and the user interface into two separate processes using TCP/IP sockets to provide the interprocess communication. As a result, the display process can run on a separate machine. This not only reduces the size of the simulation code but also implies that the window system (*e.g.*, X Windows) does not have to run on the machine performing the simulation, thereby increasing the amount of memory available for use in the simulation.



ATM Switch
4 input, 4 output Thunder and Lightning switch. Colored routing lines indicate into which buffers incoming cells can flow.

Path Display Control
Provides means to change color and persistence of displayed session path.

Simulation Control
Shows elapsed simulation time. Provides means to pause and restart simulation.

Session Path Display
Shows the session routing within the network.

Rate Meter
Indicates percentage of link rate currently in use. This rate meter is at a traffic source for the network.

Buffer Occupancy
Indicates percentage of buffer currently occupied. Each output port has one buffer for each other incoming port.

Active SessionList
Provides information about all active connections in the network.

Figure 2: Sample simulation display.

While executing the user interface on a separate machine helps reduce the computational load on the machine running the simulation, employing a client/server relationship between the display and the simulator further reduces the computational cost. The simulation does not constantly send status information to the display over the communication channel. Instead, the display requests (only) the information needed to satisfy the user demands. The simulator responds to display requests with the appropriate information, and the display process then manipulates the received data into a form suitable for display. In this way, the simulator is interrupted from normal processing only when a specific request is received; if the display is idle or not running, no processing time is wasted.

Finally, display flexibility is provided through the use of the Tcl/Tk scripting language [4]. As described in Section 3 below, almost all of the graphics-related code is written as compact Tcl/Tk scripts, which allow for rapid coding of new window types. The display uses only a small amount of C code to initiate requests and to convert the binary response data into string lists that the scripts can use.

3 Implementation Details

To provide as much flexibility as possible, the display application is implemented as multiple nearly-independent modules. Each module is responsible for the display of a particular kind of data and is implemented using both a C file and a Tcl/Tk script. Transmission of requests and reception of responses related to that display type are handled in the C file. The C code converts the simulator response into a form useful by Tcl and then invokes a Tcl procedure to perform the display action. The Tcl file implements the user interface for the data display maintained by the module.

One such display module is used to illustrate the path a simulated connection takes through the network, as shown in Figure 2. These path displays are either transient, in which case they are automatically removed from the display after a short delay,

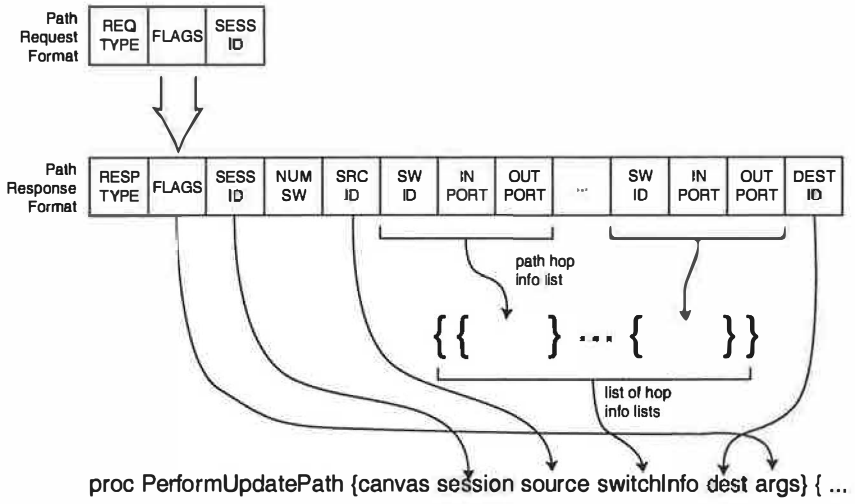


Figure 3: Mapping D_PATH response data into function call arguments.

or persistent. Because multiple routes may be displayed simultaneously, the path module automatically chooses a different color for subsequent path displays. A small control panel provides a means to change the persistence and color of displayed routes.

In the following sections, we describe both the Tcl and C code for the path module to provide a concrete example of the ease with which modules can be created.

3.1 Simulator-Display Interface

When the user wishes to display a particular piece of information, the display sends a request to the simulator. Each request consists of a single integer request-type field followed by a request-type-specific number of arguments. Figure 3 illustrates the format of the request message used to obtain session routing information from the simulator. The request-type field holds the constant integer value, **D_PATH**, assigned to path requests. Routing information requests have two arguments. The flags field argument indicates whether the path display should be persistent or transient, while the session ID field indicates the simulated connection that should be displayed.

To aid in the concurrent development of the simulator and the display, the display application is stateless with respect to communication. When the display makes a request on behalf of the user, it does not maintain any record of the request. This enables the simulator to ignore unrecognized requests, while not causing the display to hang awaiting a response that will never be received. Similarly, if the display cannot handle a response from the simulator, the response may be discarded without affecting the simulator.

Stateless communication also allows the display application to be used virtually unchanged with the real Thunder and Lightning switch. The Thunder and Lightning switch runs a small daemon process which implements the subset of simulator responses that apply within the context of a real switch. For example, the daemon responds to buffer occupancy requests, but silently ignores requests to edit or create simulation events. The interface presented to the user is the same whether the display is connected to a simulation process or a real switch.

The fact that communication is stateless, however, requires all responses from the simulator to contain sufficient information about the original request to enable the display to process the response appropriately. Figure 3 shows the format of a path response message from the simulator. Because no request state is maintained, the response includes both the session ID and the flags fields transmitted in the original request message. In fact, the simulator completely ignores the flags field when processing the request; the persistence information for the path display must be transmitted to the simulator in the request message for the sole purpose of returning it in the response message. In addition to information from the original request, the response also contains the source and destination IDs of the path, a set of triples containing information about each hop along the path, and a field indicating the length of the set. Each triple identifies the switch ID and the ports on which the session enters and exits that switch. As described later in Sections 3.2 and 3.3, the display converts these integers to Tcl lists in C code and then to display-specific alphanumeric canvas tags in order to choose the correct canvas items to highlight on the topology display.

3.2 Implementation of the Path Module in C

The purpose of a module's C code is to handle communication with the simulator over the socket. Unlike other remote display applications, such as [9], which exchange text messages over a socket, the simulator and display exchange binary data. We use XDR (external data representation) [10] routines to convert data to and from network form. This allows the display and simulator to be run on architectures with different binary representations. The initialization code, not shown here, installs a channel handler for the socket that reads raw data from the socket into an XDR buffer. The channel handler then passes the XDR buffer to the appropriate module's handler based on the response type. (Recall that simulator responses must be self-identifying because the display is stateless.)

The path module initialization code is shown in Figure 4. The initialization code simply registers a new command with the Tcl interpreter to allow a Tcl script to request a path display.

Figure 5 illustrates the code for the request procedure. We create an XDR buffer with a call to **`xdrmem_create`**, and then encode each element of the request using XDR functions. In this case, the request consists of the request type, **`D_PATH`**, a flag indicating the path's persistence, and the session ID to be displayed. When the request has been assembled, we send the entire XDR buffer to the simulator over the Tcl socket (**`PORT.socket`**) between them.

When a response is received from the simulator, the channel handler servicing the socket reads the data from the network into an XDR buffer. Based on the response type, the appropriate module's handler function will be invoked. The code in Figure 6 will be invoked for all responses of type **`D_PATH`**. The path module expects responses to be in the format indicated in Figure 3. **`HandlePath`** reads all of the data from the XDR buffer and then parses it into Tcl strings by printing into temporary character arrays. Lists are assembled as shown in Figure 3 before invoking the Tcl procedure to display the path information.

3.3 Implementation of the Path Module in Tcl

As is the case with all modules in the simulator display, all of the graphical work is done in Tcl scripts. This provides for quick development, as no recompilation is necessary when making changes to a module under development. In most cases, we can even test changes in a module on the fly, without restarting the display process, simply by re-sourcing the Tcl file.

The Tcl code for the **`PerformUpdatePath`** procedure, which displays a route on the network topology display, is shown in Figure 7. It is called by the **`HandlePath`** C function described in Section 3.2. The display's path module relies on the fact that the network topology display module is implemented in a Tk canvas window and assigns tags to all of the routing arrows on the display as follows. Each line segment associated with switch x has a tag of the form "**`switch`**x." In addition, each segment on the path leading from input port y to output port z at a switch has a tag of the form "**`in`**y**`out`**z."

As indicated in Figure 3, the **`switchInfo`** argument to **`PerformUpdatePath`** contains a list of (switch ID, input port, output port) triples. To highlight a path, we find all canvas items (line segments) tagged with both the switch tag for the appropriate switch ID and the in/out tag for the given input/output ports. Note that Tcl/Tk does not provide a direct means of locating canvas items based on two or more specified tags. We could have modified the network topology module to add an additional tag of the form "**`switch`**x**`in`**y**`out`**z" so a search could be done for a single tag. Instead, we chose to implement the **`canvasMultiMatch`** procedure to provide this missing functionality so we wouldn't need to modify the network topology module each time a search on a different group of tags was required. We then duplicate each item found and copy all attributes of the original line segment to the newly created segment, which by default is now on top of the canvas stacking order. We set the new segments to a color from a predefined color list, increase the line width, and tag the new elements with a unique tag so they can be easily distinguished later. In addition, we set a mouse binding to allow the user to change the color or persistence of the path. Finally, if the path is not persistent, we create a timer callback to delete the path after a predefined interval.

Although it is not shown here, the path module also contains a few other Tcl procedures. **`PathControlWindow`** implements a palette-like con-

```
int PathInit(Tcl_Interp *interp) {
  Tcl_CreateCommand(interp, "RequestUpdatePath", RequestUpdatePath,
    (ClientData) NULL, (Tcl_CmdDeleteProc *)NULL);
  return TCL_OK;
}
```

Figure 4: Path module initialization procedure.

```
int RequestUpdatePath(ClientData clientData, Tcl_Interp *interp,
                      int argc, char *argv[]) {
   char buffer[BUF_SIZE];
   XDR xdrs, *xdrsend = &xdrs;
   int temp, flags = 0;
   if(argc < 2 || argc > 3) {
     /* error handling omitted */
   }
   xdrmem_create(xdrsend, (void *)buffer, BUF_SIZE, XDR_ENCODE);
   temp = (int)D_PATH;
   xdr_int(xdrsend, &temp);
   if (argc > 2 && strcmp(argv[2], "persistent") == 0)
     flags | = (int)PERSISTENT;
   xdr_int(xdrsend, &flags);
   temp = atoi(argv[1]);
   xdr_int(xdrsend, &temp);
   Tcl_Write(PORT.socket, (char *)buffer, xdr_getpos(xdrsend));
   xdr_destroy(xdrsend);
   return TCL_OK;
}
```

Figure 5: C code to request path information from simulator.

trol panel interface allowing the user to change the color or persistence properties of a displayed route. Additional procedures are used to notify other modules of the posting and removal of path displays, should they wish to annotate their displays accordingly. For example, a session route displayed in black causes the corresponding entry in the session list window to be highlighted in black, as shown in Figure 2.

4 Experiences

Tcl/Tk has saved us countless hours in the development of the graphical display for the Thunder and Lightning protocol simulator. In fact, without Tcl/Tk, it is likely that the display application would never have come into existence, because our primary research involves developing protocols for high-speed networks, not designing graphical interfaces! Only the ease and rapidity of design provided by Tcl/Tk have afforded us the opportunity to develop the display interface. Nonetheless, we have uncovered several areas in the language that we believe could use further development. We present these shortcomings in this section, along with the techniques we used to circumvent them.

4.1 Sockets between Heterogeneous Platforms

When this work began, sockets were not an official part of Tcl 7.3 and Tk 3.6. It is partly for this reason that the specification for interprocess communication between simulator and display uses binary data and not strings, which would be more natural for Tcl. Nonetheless, we retrofitted both the simulator and display to use Tcl sockets when they became available in Tk 4.1. Taking advantage of the cross-platform advances in Tcl/Tk 8.0 has allowed us to port both the simulator and the display to various platforms on which Tcl is supported. Both applications should be capable of running on any Tcl-supported platform and have been tested successfully on Solaris 2.x, MacOS 8.x, and MkLinux DR3 on a PowerPC.

Despite the uniform, cross-platform access to sockets that Tcl provides, problems arise when the simulator is run on one architecture and the display is run on another. When both applications are run on the same architecture, each safely interprets incoming binary data in its native format. However, when one application is run on a little endian architecture (*e.g.*, Windows on x86 processors) and the

```
int HandlePath(XDR *xdrrecv) {
  int numSwitches, sessionID, source, dest, *switchInfo;
  int sw, item, result, i;
  int flags;
  char buffer[10];

  /* read in path flags, session ID, # of switches, source ID */
  if (!(result = xdr_int(xdrrecv, &flags)) ||
      !(result = xdr_int(xdrrecv, &sessionID)) ||
      !(result = xdr_int(xdrrecv, &numSwitches)) ||
      !(result = xdr_int(xdrrecv, &source))) {
    /* error handling omitted */
  }

  /* read in all the switch information (⟨ID, in port, out port⟩ triples) */
  switchInfo = (int *)malloc(3 * numSwitches * sizeof(int));
  if (!switchInfo) {
    /* error handling omitted */
  }
  for (i = 0; i < 3 * numSwitches; i++)
    if (!(result = xdr_int(xdrrecv, &switchInfo[i]))) {
      /* error handling omitted */
    }
  /* read in destination ID */
  if (!(result = xdr_int(xdrrecv, &dest))) {
    /* error handling omitted */
  }
  /* now we need to parse all of this information into:
   *      session, source, list of switch info lists, destination
   */
  sprintf(buffer, "%d", sessionID);
  Tcl_SetVar(gInterp, "pathTmpSession", buffer, 0);
  sprintf(buffer, "%d", source);
  Tcl_SetVar(gInterp, "pathTmpSource", buffer, 0);
  sprintf(buffer, "%d", dest);
  Tcl_SetVar(gInterp, "pathTmpDest", buffer, 0);
  Tcl_UnsetVar(gInterp, "pathTmpSwList", 0);
  for (sw = 0; sw < numSwitches; sw++) {
    Tcl_UnsetVar(gInterp, "pathTmpList", 0);
    for (item = 0; item < 3; item++) {
      /* build up switch sublist */
      sprintf(buffer, "%d", switchInfo[(sw * 3) + item]);
      Tcl_SetVar(gInterp, "pathTmpList", buffer,
                 TCL_APPEND_VALUE | TCL_LIST_ELEMENT);
    }
    /* append sublist to switch list */
    Tcl_SetVar(gInterp, "pathTmpSwList",
               Tcl_GetVar(gInterp, "pathTmpList", 0),
               TCL_APPEND_VALUE | TCL_LIST_ELEMENT);
  }
  /* Finally, invoke the update command */
  Tcl_VarEval(gInterp, "PerformUpdatePath ", " .switch.c ",
              "$pathTmpSession $pathTmpSource $pathTmpSwList $pathTmpDest ",
              (flags & PERSISTENT) ? "persistent" : "",
              (char *)NULL);
  return 1;
}
```

Figure 6: HandlePath: C code to process a Path response from the simulator.

```
proc PerformUpdatePath {canvas session source switchInfo dest args} {
  global pathTmpNum pathColors

  foreach switch $switchInfo {
    set itemList [canvasMultiMatch $canvas [list "switch[lindex $switch 0]" \
         [format "in%dout%d" [lindex $switch 1] [lindex $switch 2]]]]
    foreach item $itemList {
      #create a copy of the original, alter it, and tag it
      set newItem [eval .switch.c create [.switch.c type $item] [.switch.c coords $item]]
      foreach attribute [.switch.c itemconfigure $item] {
        .switch.c itemconfigure $newItem [lindex [lindex $attribute 0] 0] \
             [.switch.c itemcget $item [lindex [lindex $attribute 0] 0]]
      }
      set color [lindex $pathColors [expr $pathTmpNum % \
           [llength $pathColors]]]
      .switch.c itemconfigure $newItem -fill $color -width 4 -tag pathTmp$session
    }
  }
  PathPostColor $session $color
  .switch.c bind pathTmp$session <Button-3> \
       "PathControlWindow pathTmp$session $session"
  # if not persistent, schedule delete timer and store timer ID as a tag
  # so it can be cancelled later if need be
  if {[lsearch -exact $args persistent] == -1} {
    # We use eval and escape the {} so the variables will be
    # expanded now and not when the timer expires
    set timerID [eval after 5000 "\{
      catch \{$canvas delete pathTmp$session\}
      catch \{destroy .pathTmp$session\}
      catch \{eval upvar #0 pathPersistpathTmp$session pathPersist\}
      catch \{unset pathPersist\}
      PathUnpostColor $session
      \}"]
    .switch.c addtag timer$timerID withtag pathTmp$session
  }
  incr pathTmpNum
}
```

Figure 7: Tcl code to post path trace on network topology canvas.

other is run on a big endian architecture (*e.g.*, Solaris on SPARC processors), this is no longer the case. When one interprets an integer sent by the other, the receiver will "see" the integer as a different value than the sender intended because the ordering of the bytes comprising the four-byte integer are stored (and therefore transmitted) in a different order. A similar situation occurs with the Thunder and Lightning switch processor, which uses a non-standard floating point format for efficiency.

The binary socket option combined with the binary command in Tcl 8.0 provides only a partial solution to this problem. Using the binary command, integer data can be converted to a known byte ordering, and the simulator and display could each convert from the agreed upon order to their platform's native order. However, the binary command cannot be used in this way for floating point data, which is used by other display modules. In addition, the binary command would be unavailable for use within the Thunder and Lightning switch processor, as the daemon does not have access to a Tcl interpreter.

Instead, we solved this binary data representation problem by using the XDR (external data representation) standard, which imposes a common network byte ordering (and size) for all simple data types. XDR provides functions of the form **xdr_*type*** for encoding or decoding data of type *type*. The preceding section presented code examples using the function **xdr_int** to read integers from an XDR buffer. Because all data exchanged between simulator and display are converted into network form by XDR routines, we are guaranteed that there will be no "misunderstanding" between the two.

Unfortunately, XDR is not available on every Tcl-supported platform. Most UNIX variants support

```
enum xdr_direction {XDR_DECODE, XDR_ENCODE};
typedef struct xdr_struct {
    enum xdr_direction direction;
    void *buffer;
    void *next;
    int bufLen;
} XDR;

#define easyxdr(typename) \
int \
xdr_##typename(XDR *xdr, typename *value) { \
    if (!xdr->buffer) \
        return 0; \
    if ((int)xdr->next + sizeof (typename) > (int)xdr->buffer + xdr->bufLen) \
        return 0; \
    switch (xdr->direction) { \
    case XDR_ENCODE: \
        memcpy(xdr->next, (void *)value, sizeof(typename)); \
        break; \
    case XDR_DECODE: \
        memcpy((void *)value, xdr->next, sizeof(typename)); \
        break; \
    default: \
        /* error handling: unimplemented features of XDR */ \
        break; \
    } \
    xdr->next = (void *)((int)xdr->next + sizeof(typename)); \
    return 1; \
}
```

Figure 8: General template for creating XDR functions for simple data types on big endian architectures.

XDR because it is used by RPC, but the MacOS is one platform without XDR support. To enable compilation on the Macintosh platform, we have had to implement the XDR routines we use. As most simple data types are already in network form, we can simply use the C macro shown in Figure 8 to create the missing XDR functions for most datatypes. For example, the simple macro invocation **`easyxdr(long)`** will define the C function **`xdr_long`**, which converts long integers to and from network form.

Although implementing XDR functionality on big endian architectures such as the MacOS is relatively straightforward, more work is required to implement XDR on a little endian architecture. Because the typical Tcl/Tk programmer should not have to worry about such differences in a cross-platform language such as Tcl, we strongly believe that XDR should be incorporated into the Tcl language. With the introduction of sockets, Tcl has provided an elegant, uniform interface to the differing TCP/IP stacks on the various Tcl-supported architectures. The addition of XDR (or a similar network byte-order standard for all primitive data types) would provide seamless interplatform communication and thereby greatly extend the power of the Tcl socket abstraction.

There are several ways in which XDR could be incorporated into Tcl. The most obvious is an **`xdr`** Tcl command similar to the binary command added in Tcl 8.0. Encoding an integer could then be as simple as executing a command such as **`set dataToSend [xdr encode integer $myInteger]`**. Decoding a message would require reading the message into a buffer (string) and executing a command such as **`set decodedInt [xdr decode integer myBuffer]`**. Note that the **`xdr`** command must consume data from **`myBuffer`** so the variable name and not its value must be passed.

Another alternative for incorporating XDR would be to use a method similar to the stream filters provided by TclDP [5]. TclDP allows the user to register a filter mechanism with a Tcl channel such that all reads and writes for the channel first pass through the filter. TclDP filters are very flexible and work well when all data passing through the channel must be transformed in the same way (*e.g.*, uuencoding). However, XDR data are processed differently depending on the type of data the user wishes to read from the channel. Unless the filter procedure knows in advance the kind of data to be read, TclDP filters don't help with XDR. Instead, we need a mechanism that allows the user to pass a filter to each read/write call on the channel. For example, to read an XDR-encoded integer from the channel, we would call the channel read procedure, instructing it to use the integer XDR filter. This would result in exactly four bytes being read from the channel, and the returned data buffer would contain one XDR-decoded integer. Of course, Tcl would still have to provide XDR filter functions for each of the standard C types, but implementing XDR in this way would also provide the general flexibility of TclDP filters.

4.2 Canvas Improvements

Another area of Tcl that we would like to see improved is the canvas widget. Canvases are of crucial importance to our project, as they provide the most important part of our display: the network topology window. In the course of developing the topology display, we found several areas of the canvas that could be improved.

4.2.1 Relief Effects

One omission in canvas functionality is the lack of relief effects for canvas objects. This makes anything drawn on a canvas seem flat in comparison to the relief effects incorporated in the surrounding widgets.

In the case of the network topology window shown in Figure 2, we use sunken rectangles to represent the three buffers at each switch output port. To simulate this effect, we embed other canvas windows in the main canvas. This allows us to set a sunken relief effect on the embedded canvas, which gives the effect of a sunken bar graph. While this works for creating reliefed rectangles, other shapes cannot be created in this way. In addition, the complete canvas window can no longer be printed through the canvas `postscript` method because embedded windows are ignored. The generated PostScript has holes wherever an embedded window appears on the screen.

4.2.2 Selection Based on Multiple Tags

Another weakness in the canvas widget involves the handling of multiple tags. The ability to associate multiple tags with canvas items has proven invaluable in implementing the display application. As described in the previous section, tags are used to implement the path display. However, the canvas find method is only capable of dealing with one tag at a time. There are often times when items need to be selected on the basis of multiple tags. We have implemented this functionality with the **`canvasMultiMatch`** procedure shown in Figure 9, but it would be far more efficient if this functionality were added to the canvas find function or if a list command were added that returned the intersection of two or more lists.

4.2.3 Megaitems

The Tcl community has expressed a desire to have megawidget support added to Tk. Megawidgets [3, 11, 12] are new widgets created entirely with Tcl code from more basic widgets. No C programming is required to construct megawidgets, and they are indistinguishable from native widgets. A SuperWidget megawidget can be created with Tcl code such as **`SuperWidget .myWidget`** and configured with code like **`.myWidget configure -relief sunken`**. Because megawidgets have the same interface as a native widget, they can be replaced by native widgets, if they become available, without changes to the code that uses them. Although megawidget support is not yet an official part of Tk, the Tcl community has addressed this need.

We believe there is a similar need for "megaitems," the canvas equivalent of a megawidget. "Megaitems" are canvas item types created from more basic canvas item types. Like megawidgets, "megaitems" should be indistinguishable from native canvas items so they can be replaced by native canvas items should they become available.

"Megaitems" would have been invaluable in the implementation of the network topology display. Both the rate meters and the buffer occupancy bars are comprised of several native canvas items. We would have liked to have defined these to be new canvas types ("megaitems") and then used them in defining a new "megaitem" type: the ATM switch. While we did write procedures to abstract the construction of these items, it is clear from the code that they are not native canvas types, and significant sections of code would have to be changed to use a native rate meter item, were it to become available. Incorporating "megaitem" support into the Tk canvas would allow the Tcl/Tk programmer to take advantage of the benefits of object-oriented design and would make the Tk canvas more flexible.

```
#procedure to find items with multiple tags in a canvas
proc canvasMultiMatch {canvas tagList} {
    # find all items with the first tag
    set itemList [$canvas find withtag [lindex $tagList 0]]
    #delete first element (used to build initial list)
    set tagList [lreplace $tagList 0 0]

    set resultList [list]
    foreach item $itemList {
        # Get all tags for this item
        set itemTags [$canvas itemcget $item -tags]
        set isCandidate 1

        # Check to see if the item has all of the tags
        foreach curTag $tagList {
            if {[lsearch -exact $itemTags $curTag] == -1} {
                # missing one of the tags; so this item doesn't match
                set isCandidate 0
                break
            }
        }
        if {$isCandidate} {
            # the item matched all tags
            lappend resultList $item
        }
    }
    return $resultList
}
```

Figure 9: Procedure to select canvas items matching a set of tags.

5 Conclusion

In this paper we have presented the design and implementation of the graphical display program for the Thunder and Lightning network protocol simulator. The display application, unlike other Tcl/Tk applications of which we are aware, has the unique requirement that it must be implemented as a separate program running on a machine other than the computer hosting the simulation process. The display and simulator must, therefore, exchange data through the use of sockets. The fact that the display and simulator may be running on dissimilar architectures poses new challenges, despite the portability that Tcl provides. We have used code from the working display application to present the use of XDR in overcoming these inter-platform binary data representation issues. Finally, we have described several areas of Tcl/Tk that could use further development. In particular, we have identified several weaknesses in the canvas widget and described how XDR might be incorporated into Tcl. We believe that the modest effort required to include XDR functionality in Tcl would extend Tcl's socket abstraction beyond that of comparable scripting languages.

References

[1] ATM Forum Technical Committee. *ATM User-Network Interface Specification*. Number af-uni-0010.002. The ATM Forum, 1994.

[2] DARPA Thunder and Lightning Research Review. Technical report, University of California, Santa Barbara, March 1995.

[3] S. Jaeger. Mega-widgets in Tcl/Tk: Evaluation and analysis. In *Proceedings of the Third Annual Tcl/Tk Workshop*, pages 43–52, Toronto, Canada, July 1995. USENIX.

[4] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, USA, 1994.

[5] M. Perham, B. C. Smith, T. Janosi, and I. K. Lam. Redesigning Tcl-DP. In *Proceedings of the 5th Annual Tcl/Tk Workshop*, pages 49–53, Boston, MA, July 1997. USENIX.

[6] M. D. Santos, P. M. Melliar-Smith, and L. E. Moser. A protocol simulator for the Thunder and Lightning ATM network. In *Proceedings of the First Annual Conference on Emerging Technologies and Applications in Communications*, pages 28–31, Portland, OR, May 1996.

[7] M. D. Santos, P. M. Melliar-Smith, and L. E. Moser. Flow control in the high-speed Thunder and Lightning ATM network. In *Proceedings of the Seventh International Conference on Computer Communications and Networks*, Lafayette, LA, October 1998.

[8] M. D. Santos, P. M. Melliar-Smith, and L. E. Moser. A lossless, minimal latency protocol for gigabit ATM networks. In *Proceedings of the Sixth IEEE International Conference on Network Protocols*, Austin, TX, October 1998.

[9] V. Schubert. Using Tk as a remote GUI front-end for 4GL-database applications. In *Proceedings of the Fifth Annual Tcl/Tk Workshop*, pages 171–172, Boston, MA, July 1997. USENIX.

[10] Sun Microsystems. *XDR: External data representation standard.* Sun Microsystems, June 1987. RFC 1014.

[11] S. A. Uhler. In search of the perfect mega-widget. In *Proceedings of 4th Annual Tcl/Tk Workshop*, pages 125–128, Monterey, CA, July 1996. USENIX.

[12] M. L. Ulferts. [incr Widgets] an object oriented mega-widget set. In *Proceedings of the Third Annual Tcl/Tk Workshop*, pages 61–76, Toronto, Canada, July 1995. USENIX.

[13] E. A. Varvarigos and V. Sharma. An efficient reservation connection control protocol for gigabit networks. In *Proceedings of the International Symposium on Information Theory*, pages 17–22, Whistler, British Columbia, Canada, September 1995.

[14] E. A. Varvarigos and V. Sharma. The Ready-to-Go Virtual Circuit protocol: A loss-free protocol for gigabit networks with FIFO buffers. *IEEE/ACM Transactions on Networking*, 5(5):705–718, October 1997.

WinACIF: A Telecom IC Support Tool Using Tcl/Tk

David Karoly
David.Karoly@amd.com

Todd Copeland
Todd.Copeland@amd.com

David Gardner
David.Gardner@amd.com

Advanced Micro Devices, Austin, Texas

Abstract

We discuss our use of Tcl/Tk to provide software support for telecommunications Integrated Circuits (ICs). Our Windows®-based Advanced Computer Interface (WinACIF) program works in concert with reconfigurable hardware based on Field Programmable Gate Arrays (FPGAs) to provide essential coordination in laboratory data collection and analysis of a device under test. WinACIF replaces several MS-DOS® based applications. Whereas the previous implementations suffered from the classic limitations of MS-DOS, WinACIF provides the flexibility and functionality of windowing applications by virtue of its Tcl/Tk roots. Tcl/Tk not only supplies more than ample power to create WinACIF, but also adds the benefit of saving valuable time otherwise spent learning a complex API. Run-time loaded Tcl extensions provide the flexibility to support various devices having diverse interfaces. A single Tcl/Tk script dynamically builds a Graphical User Interface (GUI) based on product configuration data retrieved from a data store. Additionally, we used canvas widgets to provide an intuitive interface. For the engineer who requires control beyond that afforded by our GUI, Tcl serves as WinACIF's command language.

1. Introduction

WinACIF is a 32-bit Windows application we designed to support AMD's SLAC™ family of integrated circuits. Telecommunications equipment manufacturers use the SLAC (Subscriber Line Audio-Processing Circuit) family of ICs to build telephony linecards that interface analog telephones with digitally switched networks. WinACIF users include AMD's engineering staff and customers. Using Tcl/Tk, we created a GUI that provides a readily understood and convenient means of programming the hundreds of features contained in the SLAC IC and displaying their current state. We utilized Tcl in conjunction with our application-specific extensions (implemented in C and C++) to script interactions with the IC. The application also configures and controls an interface board that generates and coordinates digital signals between the SLAC device and PC, and between the device and test equipment (Figure 1). Tk's canvas widget provides a handy way to build a dynamically configurable diagram that depicts the signal flow in a manner familiar to the users.

2. Application Description

AMD's SLAC devices are programmable codec/filter ICs. They are essentially embedded digital signal processors that perform A/D conversion, D/A conversion, filtering, compression and expansion functions. With the advent of AMD's advanced SLAC devices, the programmability now extends to control the line power feed, ringing, signaling and test functions. This programmability benefits the linecard designer by providing the flexibility to create one hardware design that satisfies the diverse requirements of multiple markets. This flexibility, in turn, benefits the telecommunication systems manufacturer by streamlining the manufacturing process and by reducing inventory, administrative costs and time required to address new markets.

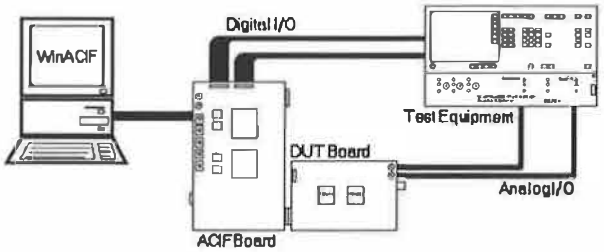


Figure 1. WinACIF lab bench scheme

The cost of all this flexibility is complexity. The most recent member of the SLAC product family has over 115 programmable command op-codes. Each op-code may be associated with a register containing up to 14 bytes of data. The data bytes themselves may contain multiple bit fields of varying widths, each controlling some programmable operating parameter of the device.

The engineer involved in the debug or application of the SLAC device faces a formidable testing and characterization task. This is particularly true when the design is intended to meet the requirements of several markets having diverse specifications. A great deal of lab bench experimentation and verification is required to ensure that the design meets the needs of all the targeted markets. WinACIF and its associated hardware provide a turn key evaluation platform that allows interactive manipulation of all the programmable features of the SLAC devices. This platform allows the engineer to easily explore the multiple "what if" scenarios inherent in linecard design. The development efforts of AMD's customers are streamlined and time to market minimized by having a tool that allows them to immediately begin evaluation of the SLAC device. They can begin the design of their application circuit and establish the proper SLAC device programming for their application without making an investment in building evaluation hardware.

WinACIF is often used in conjunction with WinSLAC. WinSLAC, a design synthesis tool, accepts the linecard design specifications and design constraints as input and produces an output text file containing the information required to configure the programmable filters of the SLAC device. WinACIF includes Tcl commands that convert this information into a stream of commands that will program the data into the SLAC device.

3. WinACIF Structure

Previously, we constructed support tools with the National Instruments' LabWindows® program under MS-DOS, adding C code to provide callback functionality. This implementation suffered from the following limitations:

- 640K of memory
- minimal functionality of widgets
- no window manager
- complex C code to create event loop and update widgets
- no scripting control mechanism
- hard-coded GUI layout
- overhead of multiple program maintenance
- difficulty of supporting new devices with multiple interface/command sets

The ability to extend Tcl with introspective commands written in C allowed us to realize our primary design goal of generalizing the ACIF software to support multiple devices having diverse interfaces.

The main Tcl script, acif.tcl, loads the core of the application, acif.dll. Acif.dll includes a data store comprised of arrays of C structures that contain the programming command sets for each of the SLAC devices. After loading acif.dll, the Tcl script uses one of the new commands, acifProducts, to ask the data store to return a list of available products. The Tcl script uses this list to construct an array of radiobuttons from which the user selects a product name. Upon selection acifProducts is executed again, this time with the chosen name as an argument. This execution configures pointers in the underlying C code, so that subsequent operations reference the appropriate section of the data store.

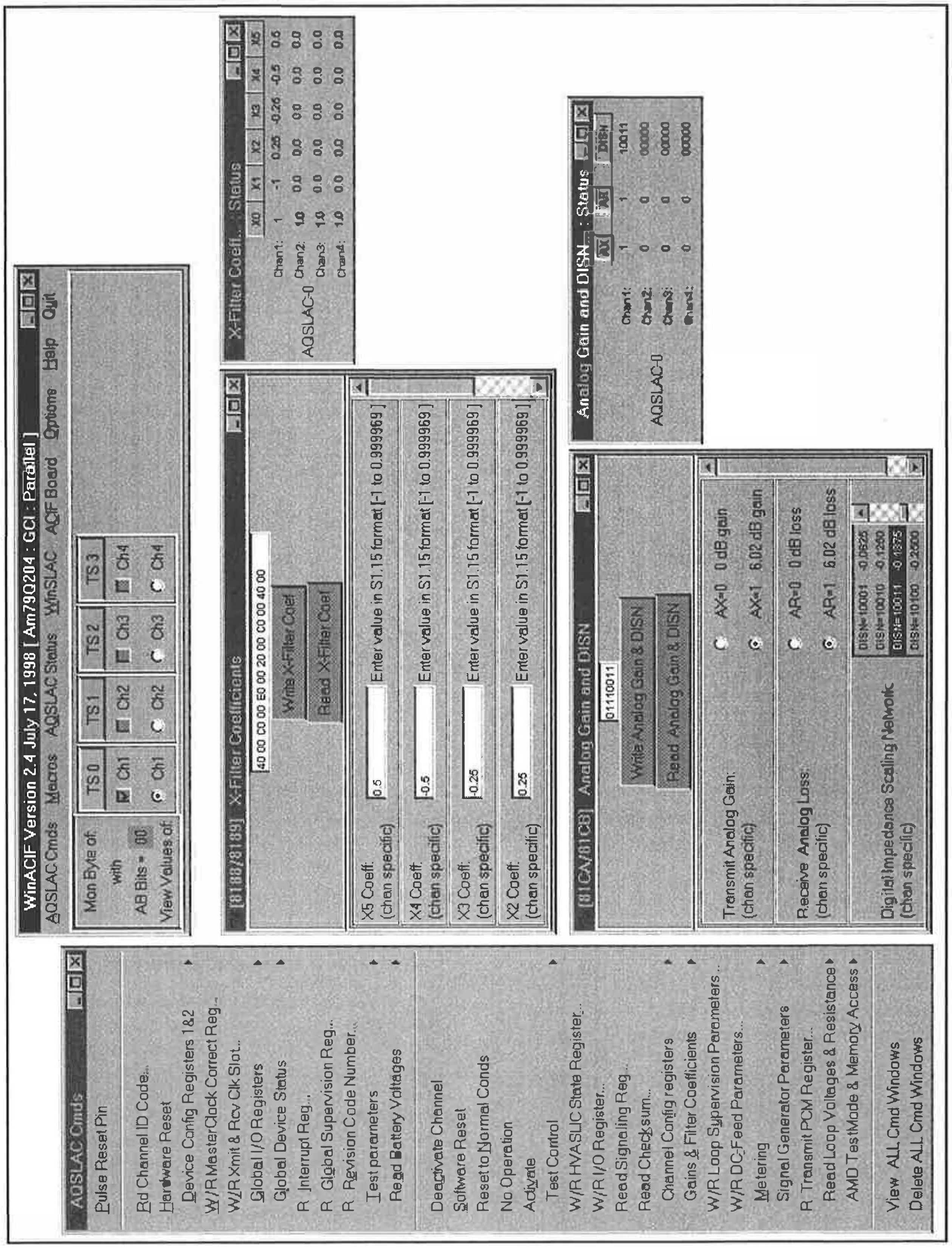
Throughout the program, we used this strategy of dynamically creating the GUI based on data obtained from the underlying data store. Depending on the argument sent with the command, our Tcl commands either return configuration information from the data store, or perform some related operation, such as writing or reading from the SLAC device. This structure made possible the writing of generic Tcl code that created different GUIs depending on the information in the data store.

In addition to acif.dll, the application may load one of several other DLLs, depending on the options selected. For example, the SLAC devices have a variety of hardware interfaces. Some new devices even offer the choice between two interfaces. Downloading a different configuration file into the FPGAs (Field Programmable Gate Arrays) on the ACIF board redefines the interface logic. Acif.dll reconfigures the software by unloading the old driver DLL and loading the one that supports the new configuration. The new driver DLL defines new Tcl commands that control the interface logic implemented in the FPGAs.

The various DLLs and the application-specific Tcl commands they provide are summarized in Appendix A.

4. SLAC IC Command Windows

To make our users as comfortable as possible with the new WinACIF tool, we designed our GUI to resemble the device data sheets with which they are so familiar. After selecting a product name, the user views WinACIF's main menu, which contains eight menu items. The first item contains the SLAC commands. When the user selects this item, a pull-down menu appears listing command descriptions for programming the selected SLAC device. If the user chooses a command with no associated register, WinACIF will execute the instruction directly upon selection from this menu. Otherwise, if the user selects a command that reads from or writes to a register, it will launch a top level window where the user can specify or view the data parameters in the associated register.



Mon Byte of: TS 0 [✓ Ch1] Ch2 Ch3 Ch4, AB Bits = 00; View Values of: [✓ Ch1] Ch2 Ch3 Ch4

40 00 C0 00 E0 00 20 00 C0 00 40 00

X5 Coeff: (chan specific)	0.5	Enter value in S1.15 format [-1 to 0.999969]
X4 Coeff: (chan specific)	-0.5	Enter value in S1.15 format [-1 to 0.999969]
X3 Coeff: (chan specific)	-0.25	Enter value in S1.15 format [-1 to 0.999969]
X2 Coeff: (chan specific)	0.25	Enter value in S1.15 format [-1 to 0.999969]

01110011

Transmit Analog Gain: (chan specific) AX=0 0 dB gain [✓ AX=1 6.02 dB gain]

Receive Analog Loss: (chan specific) AR=0 0 dB loss [✓ AR=1 6.02 dB loss]

Digital Impedance Scaling Network: (chan specific)

```
DISN=10001  -0.0625
DISN=10010  -0.1250
DISN=10011  -0.1875
DISN=10100  -0.2500
```

		X0	X1	X2	X3	X4	X5
AQSLAC-0	Chan1:	1	-1	0.25	-0.25	-0.5	0.5
	Chan2:	1.0	0.0	0.0	0.0	0.0	0.0
	Chan3:	1.0	0.0	0.0	0.0	0.0	0.0
	Chan4:	1.0	0.0	0.0	0.0	0.0	0.0

		AX	AR	DISN
AQSLAC-0	Chan1:	1	1	10011
	Chan2:	0	0	00000
	Chan3:	0	0	00000
	Chan4:	0	0	00000

Figure 2. Desktop showing main window (top center), tear-off command menu (left), two command windows (center) and two status windows (right).

When the user has configured the desired combination of parameter values using the command window, they can write the corresponding data bytes to the SLAC device by clicking the write command button. We divided the command window into three main sections:

- An entry widget that displays the data in either binary or hexadecimal form. The user has the option of specifying the data byte by typing into this entry widget.
- Two command buttons, each sending a particular command op-code to the SLAC device. Typically these buttons correspond to read and write operations.
- A series of widgets on a scrollable frame that display the parameters programmed by the data bytes and allow individual modification of each. The scrollable frame, created with the canvas widget, makes it possible for the user to display the parameter of interest, even if the window is resized to be smaller. The user may thereby optimize their use of screen real estate.

Figure 2 shows the main window with the pulldown menu torn off, two of the possible command windows open on the desktop, and two status windows. We will discuss status windows in section 5.

Some other development tool kits require that an image be manually created for each window in the application. Instead, we wrote a data driven Tcl procedure that is used to build all of the command windows. An introspective mechanism queries the data store to determine the desired data format, the widget types, and all other information to place in the window. Since the data store defines hundreds of variations on the basic window, we avoid an enormous development and maintenance burden through this approach. Should a need arise to modify the presentation of the command windows, the edits to the one Tcl procedure will automatically apply to all of the command windows for all the products supported on WinACIF.

The write command button executes the acifDut command with arguments defining the operation and the data bytes. The acifDut command abstracts all the details of communicating with the hardware and the maintenance of the data structures. The first argument to acifDut is a command mnemonic that identifies the operation. The second argument is the user-specified data bytes. The C code that implements acifDut uses Tcl's hash table facility to map the mnemonic, the first argument, to a pointer referencing the associated information within the data store. This provides the hexadecimal op-code as well as information needed to validate the data bytes, and to parse them into bit fields corresponding to the various parameters.

5. Maintaining and Displaying SLAC Device State

Interacting with a device with over 700 programmable parameters would be impossible without a means of recalling and displaying the current settings. The ability of some SLAC devices to support multiple telephone channels further complicates the tracking of the configuration variables. Some parameters have a channel-specific scope, such as the gain of an individual channel, while others have a global scope within the device, such as the choice of an incoming clock frequency. The data store contains information concerning the scope of each parameter.

When commands are sent to the SLAC device, the C code responsible for interfacing to the hardware also updates a Tcl array called DUT (Device Under Test). This array records the values of all of the SLAC device parameters programmed by the user through the command windows. DUT's element names correspond to the various programmable parameters of the SLAC device. We made DUT read-only to insure that the content accurately reflects the state of the SLAC device. The Tcl script cannot alter the contents of DUT except by executing the command which programs the SLAC device. We store the parameter values in the array DUT in a format meaningful to the user. To convert the bit-field values used by the device into an understandable format, such as decibels, amperes, ohms, etc., the C code uses data supplied by the data store.

Using this strategy, we can easily create a display of the current device configuration. We merely set up a label widget that displays the desired elements from the DUT array. The main menu of WinACIF provides the user with the ability to view these status windows on the desktop (Figure 2). These windows show the current value of a parameter or values of a group of related parameters across all of the channels in the device.

6. Variable Traces and Bindings

In the implementation of the command window, Tcl's variable trace feature provides a convenient mechanism for maintaining the correct relationships between the widgets and the underlying data structures. When the window is created, temporary variables are created for the parameter modification widgets in the bottom section of the window. These variables are initialized with the appropriate parameter values from the DUT array. The temporary variables allow the user to setup any combination of changes to the parameter values, independent of the actual, read-only values stored in the DUT array. Variable traces on the temporary variables trigger the execution of code that maintains the data byte display in the top of the window. A Tcl command extension is called that uses information from the data store to convert the set of parameter values into their corresponding bit-fields and assemble these bit fields to create the data bytes.

A user may also type into the data byte entry, thereby triggering an update of the parameter values displayed in the widgets below. The user may then write the new data bytes to the SLAC device by clicking on the Write button. This action calls a Tcl command extension that programs the SLAC device and updates the values in the DUT array to reflect the new programming of the device.

As discussed in a later section, the user can also program the SLAC device independent of the command window by executing a script containing the acifDut command. In this case, the values displayed in any affected command window must be updated. Variable traces placed on the appropriate elements of the DUT array achieve this effect. When the acifDut command changes the value in the DUT array, the trace fires, executing code that copies the value into the corresponding temporary variable. This updates the widgets and also fires a trace that executes the code that updates the data byte entry.

We rely on Tcl's variable trace mechanism to maintain the correct relationships among the DUT array, the parameter widgets, and the data byte display. The event-driven nature of this code makes good documentation essential.

7. Graphical Board Control

Tcl's canvas widget provided us the opportunity to represent signal flow in a graphical format that is intuitive to our users. The SLAC devices may use one of several different interfaces to transmit data. WinACIF depicts a particular interface by drawing a graphical representation of the signal flow and switching arrangements on a canvas widget. Figure 3 shows one of the possible interfaces. This figure depicts the SLAC device transmitting and receiving a serial bit stream that is logically grouped into 8 bytes. The bytes labeled B1 and B2 consist of digitized audio data that can be captured for conversion to parallel form and routed to or from a piece of telecom test equipment. The window provides interactive control over the evaluation board hardware that performs the signal routing and conversion.

Rather than embedding radio or checkbutton widgets in the canvas, we tagged groups of lines and assigned bindings to the tags to reconfigure the evaluation board. For example, the row of arrows pointing up from the top row of bytes works together to comprise a one-of-four selector switch. Clicking on one of these arrows changes the selection by executing an AMD-created Tcl command that programs the appropriate hardware register. The Tcl command also updates a read-only array, the acifBoard array, that tracks board state. The black arrow indicates the current choice.

The Tcl code executed by the binding event can be as elaborate as needed to achieve the desired effect. In the bottom row of bytes in Figure 3, each of the B1 and B2 boxes has two arrows feeding into it. These arrows comprise a "one, the other or neither" selector switch. These kinds of switch arrangements make intuitive sense to the engineers who use WinACIF.

The need to write a program to draw on the canvas rather than using an interactive "what-you-see-is-what-you-get" drawing tool often dismays newcomers to Tcl/Tk. In reality, we find defining the graphic programmatically is an asset. The interface depicted in Figure 3 has an alternate mode of operation that transmits 16 rather than 8 bytes. The Tcl procedure draws the graphic based on the number of bytes passed as an argument. This approach makes it easy to redraw a different arrangement.

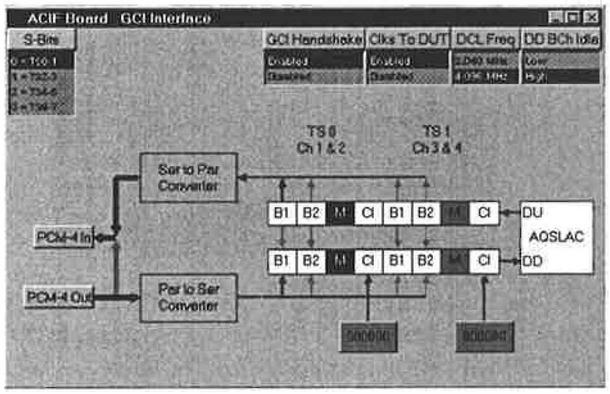


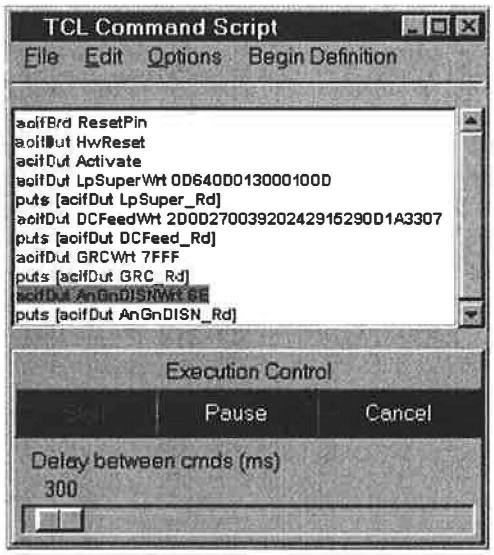
Figure 3. Board programming window

8. User Scripting: More Than a Macro

The user interactions with WinACIF discussed so far all operate directly on the programming of the SLAC device or on the ACIF hardware. Testing of a particular linecard configuration may require the programming of hundreds of commands. The window in Figure 4 provides the ability to reproduce this set of operations at a later time without laboriously repeating the exact sequence of mouse clicks and button presses.

As the user interacts with the other windows of the GUI, this window can capture the Tcl commands that program the SLAC device and the ACIF board. The user does not need to know the command names or how to specify their arguments. Note, however, the command names and arguments have been designed to be self-explanatory to users familiar with the SLAC device.

We based the Command Script Window on the text widget. The user can modify the commands using cut and paste operations or by placing the insertion cursor into the middle of the sequence and inserting additional commands by interacting with the other windows of the GUI. The user can replay the script from this window with a user variable delay between commands and insert breakpoints into the script. We provided an option that repetitively sends the commands in a loop useful for debugging certain aspects of SLAC device operation. The user may save and load these command sequences as files.



```
acifBrd ResetPin
acifDut HwReset
acifDut Activate
acifDut LpSuperWrt 0D6400013000100D
puts [acifDut LpSuper_Rd]
acifDut DCFeedWrt 2000270039202429162900 1A3307
puts [acifDut DCFeed_Rd]
acifDut GRCWrt 7FFF
puts [acifDut GRC_Rd]
acifDut AnGnDISNWrt 6E
puts [acifDut AnGnDISN_Rd]
```

Figure 4. Command script window

The above describes more than a macro and replay mechanism because the user is saving an independent Tcl script. For the engineer who requires more sophisticated control, this provides an avenue for writing Tcl scripts that can completely automate testing processes. Certain testing scenarios may require looping, conditional branching, or file IO. He can use the full power of the Tcl interpreter to solve these problems. Tcl's simple syntax, which in many areas is similar to C, reassures the users that the learning curve will not take too much time away from already busy schedules. The Tcl extensions implemented in WinACIF abstract the details of communicating with the SLAC device and the ACIF hardware. Each engineer can now craft customized scripts, share those scripts with others, and even submit a valuable addition for incorporation into WinACIF.

9. Conclusions

The port of Tcl/Tk to the Windows' platform greatly accelerated the evolution of the WinACIF software from a MS-DOS to a Windows application. The difficulty and time required to learn the basic skills to be a proficient Windows developer has barred most engineers from developing their own GUI-based applications. The conventional toolkits make GUI development a field for software specialists. Charles Petzold says it best,

> "...please do not fear that you are missing some vital part of your brain that is required to be a successful Windows programmer. This initial confusion is normal, and don't let anyone tell you differently. Windows-based programming is strange. It's weird, it's warped, it's awkward, it's convoluted, it's mind-boggling..."[1]

We find Tcl/Tk to be an empowering tool which bypasses most of these difficulties thereby drastically shortening the learning curve and the development time required to build GUI-based applications. For engineers whose focus has been the application domain, Tcl/Tk's ease makes Windows development possible.

By defining the GUI programmatically using Tcl/Tk script, we structured this application in a more powerful and flexible manner. The ability to extend Tcl with C code allows implementation of complicated data structures and interfaces to hardware. Dynamic reconfiguration for various products and interfaces could then be implemented, consolidating support for the entire product family in one program. The application is readily extendable to provide support for future products. Additionally, Tk's canvas widget provides a powerful tool for delivering information in a graphical form familiar to our users.

10. References

[1] Petzold, Charles. *Programming Windows 95: The Definitive Developers Guide to the Windows 95 API.* Microsoft Press, 1996. ISBN 1-55615-676-6.

Appendix A : Tcl Command Extensions used in WinACIF

ACIF.DLL

defines the following Tcl command extensions related to programming the SLAC device:

AcifProducts	Returns information describing the SLAC products and interfaces that are supported by WinACIF. Used to select the product/interface and configure WinACIF accordingly.
acifDut	Returns information describing the command set of the SLAC device. Writes or reads SLAC device control interface and updates Tcl array DUT which tracks the state of the SLAC device.
AcifDutEntryCheck	Checks validity of data byte string for proper number of binary or hexadecimal digits.
AcifDutEntryFromVals	Converts parameter values to bits and combines them to create data byte string.
AcifDutEntryToNewVals	Fractures data byte string into parameter values and updates temporary variables for command window widgets.
AcifDutEntryToHex	Converts value to hex form without spaces.
AcifParmEntryCheck	Validates the data when an entry widget is used to enter a parameter value.
acifExit	Cleans up DLLs and exits.
acifDown	Loads the proper DLL for downloading a WinSLAC file.
AcifSelectPort	Returns the list of available communication ports. Selects a port to use.
AcifFileStatus	Returns the compilation date and time of WinACIF's currently loaded DLLs.
AcifBinToVal	Converts a string of binary digits to a specified decimal format.
AcifValToBin	Converts a decimal value to a string of binary digits according to a specified format.
PAUSE	Executes pause when encountered in a user-defined script.

PCM.DLL

defines the following Tcl command extensions related to controlling the interface board when implementing a PCM/MPI interface:

acifBrd *option value*	Programs *value* into the ACIF board hardware. Updates the corresponding element of the acifBoard associative array. *Option* identifies one of many registers contained in the FPGA logic. Each of these registers controls some aspect of the operation of a PCM/MPI interface.

GCI.DLL

defines the following Tcl command extensions related to controlling the interface board when implementing a GCI interface:

acifBrd *option value*	Programs *value* into the ACIF board hardware. Updates the corresponding element of the acifBoard associative array. *Option* identifies one of many registers contained in the FPGA logic. Each of these registers controls some aspect of the operation of a GCI interface.

Charity Telethon Supported by Tcl/Tk

Dave Griffin
Compaq Computer Corporation

Abstract

A set of Tcl/Tk Version 8 scripts supports a telethon that raises money for needy families. This paper describes the system and how it was constructed. Because the system uses nearly every major feature of the Tcl and Tk system, including the Tcl web server, this paper serves as an example that emphasizes the power of universal scripting in both the development and deployment of a distributed system.

Introduction

For the past 19 years in Maynard, Massachusetts, the local high school radio and television station, WAVM, stages a telethon to raise money for a local charity supporting needy families in the area. This annual community event culminates in a 40-hour, non-stop television telethon, which includes live auction segments taking place throughout the event. The telethon weekend is staffed by over 100 students (from 6th through 12th grade), parents, and other community members.

In 1992 a (then) new network and VAX/VMS computer system had been installed at the high school where the telethon originates. The author wrote the first telethon auction support program in some earlier scripting languages: DCL (Digital Command Language) and GNU awk. This program served the telethon well, but was an absolute nightmare to maintain. It was also dependent on hardware (VAX 4000 and VT420 "dumb" terminals) that has a limited future within the school system.

For the 1997 WAVM Telethon, I decided that it was time to free ourselves from the trusty VMS-based system and move to a system that took advantage of the PCs, servers, and network infrastructure that has grown over the past 5 years. WAVM's Web department was also growing rapidly and, for the first time, taking a major role in the telethon's presentation to the public. So, we were interested in integrating facets of the auction to an unknown Internet audience.

I was prepared to develop the whole system myself, but a fellow Tcl hacker wanted to learn more about Tk and offered to help. However, his time was limited. This situation was further complicated by his taking a new job at another company–which meant that our communication would be primarily through e-mail. We opted for a simple-but-flexible architecture that takes advantage of the computing infrastructure at the school, as shown in the following diagram:

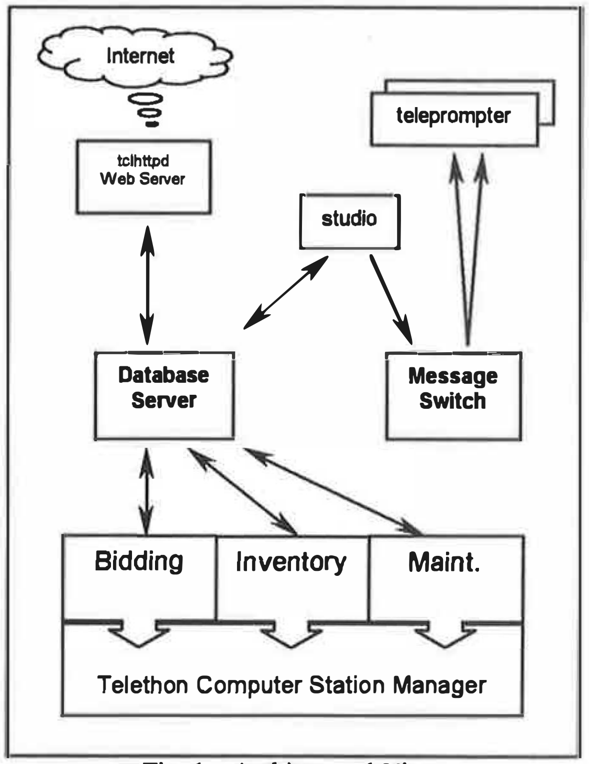


Fig. 1 – Architectural View

Even a "midnight" project needs priorities and we set them early on, as follows:

1. Support the auction. Provide programs to maintain the inventory of donated items and accept bids made over telephones during the telethon.

2. Keep the studios in touch with the auction. The first priority here was a way of letting the telethon hosts know the current top bids.

3. Make managing the telethon easier on the studio floor managers and on the adult support team. This includes reports, real-time monitors, etc.

4. Integrate the World-Wide Web with telethon activities--particularly the auction. (In parallel with all this work, we were preparing to webcast the full 40-hour telethon live via RealVideo.)

The Database Server

The first component to be constructed was the database server. I initially considered using the AltaVista Forum Toolkit [1], because it had a built-in database manager with journalling, and I was also very familiar with the interface. I did not consider other database packages (for example, MySQL) because I had no prior experience with them (and I didn't really have time to evaluate and select one). Plus it would require compilation and testing for my target platforms (Intel & Alpha). The AltaVista Forum software is based on Tcl 7.6–another minus. I opted to use a 100% Tcl approach, using the Tcl 8.0 release to construct a simple in-memory database with journalling for reliability [2].

The auction database consisted of three datasets: the items, the callers, and the bids. Each dataset had unique characteristics in terms of access and processing, but they all required common storage management and journalling/recovery features. (A windstorm in 1993 knocked out power during the middle of a telethon, so reliability features were historically mandated.)

Each dataset was given its own Tcl namespace within which to work. Database operations such as Add, Remove, etc., were implemented as Tcl procedures within the namespace. Because the total estimated size of the database was under a megabyte, the data for each dataset was stored in memory (as a Tcl array or list). This decision provided the server with a very high-performance base on which the reliability features were built.

It is interesting to note that while the Items and Callers datasets were implemented as one or more arrays that provided keyed access to the records, the Bids dataset was implemented as a simple Tcl list. Tcl 8 list operations are orders of magnitude more efficient than previous versions, and accessing elements of a list with 1000+ members consumes very little processing time. Based on historical data from the old system, we calculated an upper limit of 2,000 elements for any one list or array. So, we were well within what Tcl could efficiently manage using this simple approach to databases.

A single set of Tcl procedures implemented common operations such as Open, Close, Create Checkpoint, and Recover. These procedures were entered into each dataset's namespace (**eval**), where they merged with the other dataset-specific operations. This technique simulates an object-oriented inheritance operation, greatly reducing the amount of code needed for each dataset. The following figure shows all of these common methods as implemented in the database server:

```
Set CommonDBMethods {

  proc Checkpoint {} {
    variable dbInfo
    variable fileBase
    variable tlogFile
    close $tlogFile
    incr dbInfo(version)
    Save $fileBase.check$dbInfo(version)
    set_file $fileBase.tlog$dbInfo(version)
""
    set tlogFile [open $file-
Base.tlog$dbInfo(version) a]
    set_file $fileBase.info [array get
dbInfo]
  }

  proc Open { dbname } {
    variable fileBase
    variable tlogFile
    variable dbInfo
    variable inRecovery
    set inRecovery 0
    set fileBase $dbname
    array set dbInfo [get_file $dbname.info]
    Load $dbname.check$dbInfo(version)
    set tlogFile [open
$dbname.tlog$dbInfo(version) a+]
    if {[file size
$dbname.tlog$dbInfo(version)] > 0} {
        Recover
        Checkpoint
    }
  }

  proc Create { dbname } {
    variable fileBase
    set fileBase $dbname
    set_file $dbname.info "version 1"
    set_file $dbname.check1 ""
    set_file $dbname.tlog1 ""
    Open $dbname
  }
  proc Close {} {
    variable tlogFile
    Checkpoint
    close $tlogFile
  }

  proc TLog { tx } {
    variable tlogFile
    variable inRecovery
    if {$inRecovery} return
    puts $tlogFile $tx
    flush $tlogFile
  }

  proc Recover { } {
    variable fileBase
    variable tlogFile
    variable dbInfo
    variable inRecovery
    log recover "Recovering a database..."
    set inRecovery 1
    seek $tlogFile 0 start
    set tx ""
    while {[gets $tlogFile tLine] >= 0} {
      append tx $tLine
      if {![info complete $tx]} {
         append tx "\n"
         continue
      }
      log recover "/$tx/"
      eval $tx
      set tx ""
    }
    set inRecovery 0
  }
}
```

The common methods maintain a so-called `.info` file for each dataset, which coordinates the versioning of journal and checkpoint files.

The journalling and recovery feature exploits the "data represented as scripts" philosophy [3]. The lowest level database operations are implemented as Tcl procedures. The procedures that modify the database simply "record themselves" in the journal file as a Tcl script, as in the following example:

```
proc Items::SetRaw { key data } {
    variable items
    variable topKey
    set items($key) $data
    TLog [list SetRaw $key $data]
    if {$key > $topKey} {set topKey $key}
    return ""
}
```

When the database server shuts down, each data set will save a copy of itself to disk (checkpoint). This is as simple as `puts -nonewline $chan [array get items]`. When a database is opened, the latest checkpoint file is read into memory. Any transactions in the journal are then replayed by reading in the procedure calls and evaluating them, thus reapplying the missing transactions.

A fourth namespace, called "Tx", holds all the public transaction methods, again implemented as Tcl procedures. A Tcl **socket** connection manager accepts incoming TCP/IP connections. Upon receipt of a connection it then sets up a **fileevent** with the `Tx::tx` procedure called whenever the channel is readable. The database server implements a very simple protocol that allows the movement of arbitrary amounts of data. For both requests and responses, the data length is sent first (terminated by a linefeed), followed by the data.

The entire database server is implemented as an 850-line Tk script, which also provides a logging window for database events, a checkbox to enable debugging instrumentation, and a master switch to enable or disable bidding. The latter is used to assure that none of the bidding stations can accept bids until the auction officially starts. (The author could be seen running across the room in a panic at the start of the auction, because the first phone call had arrived but this "master switch" had not yet been thrown.)

The Auction

An auction is divided into 3 phases and there is specific software devoted to each phase.

Prior to the 40-hour telethon event, the focus is on entering new auction items into the database and tuning up the caller database; we developed some Tk scripts to handle these activities.

During the auction three to five "bidding stations" are active. These stations are manned by students who answer phone calls and process the bids. Special care was taken in the design of the bidding station's Tk interface to keep the flow of questions and responses easy to manage with a minimum of typing. This is particularly important when you have a phone in the one hand and the knowledge that many people are trying to call it at the same time. Care was also taken to allow backtracking and quick resetting in case the caller (or the student at the station) "got lost."

A number of Tk scripts support the auction by allowing the auction management team the ability to keep track of the bidding, providing statistics to the telethon managers, and opening and closing bidding on individual or groups of auction items. The management scripts were fine-tuned during the telethon (one or two reporting scripts were actually written during lulls in the action).

After the telethon signs off the air, a team of adults and students use the various parts of the system to reconcile the paper and computer-based records to designate winners for all items.

All of these scripts were under the control of an "umbrella" script, which did a simple password check and gave access to a subset of the capabilities as needed. This script would dynamically load (**source**) in the particular program needed from a master network file share.

Supporting the Studios

Two television studios used during the telethon. We needed a way of relaying current bid information to the on-camera telethon hosts in a smooth and inobtrusive manner regardless of which studio they might be in. During the course of developing this capability it became apparent that a powerful and flexible system could easily be constructed to provide this critical information and potentially much more.

A "message switch" Tk script was designed to run during the entire telethon, much the same way the database server was always available. The message switch would accept messages via the standard **socket** and **fileevent** mechanisms. These messages could either listen for messages or leave a message. Every time a message was left, its payload would be the response for any outstanding `listen` messages. This was done by the `listen` transactions doing a **vwait** on the message number, which was incremented by the `leave` message. The following procs implement the core functions of the message switch:

```
proc connHandler { chan ipAddr port } {
    global activeChannels
    log conn "Starting channel $chan"
    fconfigure $chan -blocking no
    fileevent $chan readable "request $chan"
    set activeChannels($chan) "open"
}

proc request { chan } {
    global activeChannels
    fconfigure $chan -blocking no
    if {[gets $chan rqst] < 0} {
        fileevent $chan readable ""
        close $chan
        log conn "Closing channel $chan"
        unset activeChannels($chan)
        return
    }
    log debug "Tx $chan: $rqst"
    # Execute transaction (unsafely)
    if {[catch {eval $rqst} err]} {
        response $chan [list ERROR $err]
    }
    catch {flush $chan}
}

proc response { chan data } {
    if {[catch {puts $chan $data} err]} {
        log debug "Response Error: $chan"
    } else {
        log debug "Sent $chan data: $data"
    }
}


proc listen {} {
    upvar chan chan
    global messageNumber
    global lastMessage
    vwait messageNumber
    response $chan $lastMessage
}

proc leave { text } {
    upvar chan chan
    global messageNumber
    global lastMessage
    set lastMessage $text
    incr messageNumber
    response $chan \
        "Message $messageNumber sent"
}
```

The mechanism was very simple and did not attempt to provide any guarantees regarding message delivery, but it was more than sufficient for the task.

Two other scripts were written to communicate with the message switch. The first script acted as a transmitter, which could relay arbitrary text from the keyboard to the switch, or it could perform a simple database lookup (to the database server) and send the results to the message switch.

The final script was the Tk "teleprompter." Copies of this script were run in the two television studios with the computer monitors placed on top of the television "talent" monitors. Each teleprompter had a pull-down menu which allowed it to designate where it was located. This allowed it to screen out messages that were not intended for it (the transmitter scripts could optionally designate where it wanted the messages to appear). Using very large fonts, the teleprompter script displayed the current bids and any other text messages for the on-camera students. Once the PC was set up with this script, it was left to run unattended for the duration of the telethon.

The transmitter scripts were run on a few computers in the studio area and in the main control room. These were staffed by students allowing the on-camera students to call out the number of an auction item that was being discussed and have the latest bid information immediately available on the teleprompter.

When no messages were received by the teleprompter for approximately a minute, an **after** event would display a "countdown clock" of how many hours were left in the telethon.

Bringing the Auction to the Web

Although providing a Web interface was desirable, it was pretty much last on the priority list. Fortunately, I had the opportunity to be using the Tcl web server [4] in a separate project, so it appeared that integrating an auction inventory query facility would be relatively straightforward.

A "Telethon" package was constructed that provided the code for a small set of URL mappings. Like the database server, a separate namespace was established to hold the Telethon package's context (e.g., a shared TCP/IP connection to the database server). It's primary job was to issue database transactions and to format the results in HTML.

The ability to look up individual items, or get reports on multiple items turned out to be convenient for everyone. Because Tcl/Tk's print support is a bit weak, we made our reports available via the web server (or generated them in HTML), and let the browsers do the formatting and printing for us.

An interface to the message switch was also created to allow Internet viewers to interact with the students. However, there was insufficient time to fully develop and test the capability. So, we dropped back to using e-mail for feedback.

Results: Trial By Fire

The telethon would eventually utilize at least thirteen computers deployed in various rooms and studios. Given our limited resources and fixed schedule there was not much time for testing the system in the "all up" configuration. Critical components like the database server were unit tested by test harness scripts, to flush out any bugs and to validate that the performance was acceptable.

Much to our delight, the software performed nearly flawlessly. A few glitches in the bidding station program were corrected on the fly during a lull in the auction. The database and message servers ran non-stop and easily handled the load placed upon them. Nearly 500 items were entered into the auction inventory, and we accepted about 1,500 bids.

During the 40-hour telethon, the database server processed 25,213 transactions. The message switch routed 2,848 transactions to the various stations in the studios.

Our custom web server was an unexpected bright spot. We really were not sure if the idea of using the Internet for a local event such as ours would attract anyone. During the telethon, just over 7,800 web transactions were processed, with a number of small updates to the program applied on-the-fly and no interruptions in service. Feedback received during the auction was universally positive for this service, and we reached people far outside the normal broadcast range. For the short amount of time it took to make it all work, this was well worth the effort.

Construction Notes

The true "script once, execute anywhere" was critical to the timely development of the software. This is highlighted by two observations:

1. The author has no information about the platform on which his partner developed and tested the bidding program. The scripts arrived as attachments to e-mail messages, which were saved to the testing area and used immediately.

2. While the systems used for actual telethon operations varied between Intel and Alpha, and between Windows NT and Windows 95, a major portion of the development of this distributed system was done while in the passenger seat of the family car on a standalone Win95 laptop. Also, portions of the custom web server were tested on a Digital UNIX system.

We never had to think twice about whether or not the scripts would behave properly when they ran on the different platforms. The only difference that had to be accounted for was font-size specifications required by different screen resolutions. This problem was taken care of by implementing a pull-down menu of base font-size preferences on the programs where it mattered.

Because all of the auction stations accessed the scripts via a network file share, on-the-fly bug fixes and updates were immediately available to the simply by closing and re-opening the appropriate window and letting the umbrella script re-source the application. At least one important bug fix was "distributed" to the stations in between phone calls.

Coding started almost exactly one month before the telethon. It took roughly 40-60 hours of work to design, code, and test the system up to the time the telethon went on the air. Another 2-3 hours of work was invested during and after the telethon to handle unanticipated issues.

Reuse of existing scripts and code, such as the Tcl web server, made it possible to add new features quickly as they became necessary. The decision to build the database server "from scratch" rather than to use a more traditional database server did not adversely affect the project. An analysis of the database server shows that roughly 325 lines of code actually implement the core database functions; the remaining code would have been necessary even if an SQL database was used instead.

Summary

Our telethon raised over $36,700 for local needy families, with the auction bringing in nearly $9,600, both amounts were new all-time records.

Tcl/Tk was instrumental in providing a modern, robust system on which future telethons will build. Tcl 8.0's performance, customizable web server, and platform independence were key assets in allowing an organization without significant resources to construct a distributed auction manager.

With the success of the telethon behind us, we are exploring how to apply this technology to day-to-day station operations. Our first task was to create a way to switch our webcasting services on and off via the web server. For upcoming telethons we will explore increasing the interaction between the web audience and the studios (messages, challenges, etc.), perhaps with limited auctioning occurring on the web itself (without SSL we need to find a way of easily authenticating bidders). One thing is for certain, the WAVM Telethon will continue to be "Tcl Powered".

References

[1] David Griffin. "Tcl in AltaVista Forum", 5th Annual Tcl/Tk Workshop Proceedings, 1997

[2] Andrew Birrell, Michael Jones, and Edward Wobber. "A Simple and Efficient Implementation for Small Databases", Digital SRC Research Report 24, 1988

[3] John Ousterhout. "Tcl and the Tk Toolkit", Section 28.4., Addison-Wesley 1994

[4] Brent Welch and Steven Uhler. "Web Enabling Applications", 5th Annual Tcl/Tk Workshop Proceedings, 1997

The Tycho Slate: Complex Drawing and Editing in Tcl/Tk

H. John Reekie and Edward A. Lee
School of Electrical Engineering and Computer Sciences
University of California – Berkeley
Berkeley CA 94720
{johnr, eal}@eecs.berkeley.edu

Abstract

This paper introduces the Slate package, which has been developed as part of the Tycho project at UC Berkeley. The Slate is layered over the Tcl/Tk canvas, and contains features that we believe to be useful for implementing complex graphical editing and visualization widgets. The first key feature is the ability to define new item types in Tcl. The second is an implementation of the concept of interactor, *which abstracts low-level mouse events into self-contained objects. The third is access to and modification of items based on their shape, rather than raw coordinates. Combined with a straight-forward implementation of the model-view-controller architecture, the Slate is capable of implementing quite sophisticated graphical editors.*

1 Introduction

The Tk canvas provides a simple but powerful set of structured graphics primitives, and is perhaps the easiest toolkit available for drawing simple 2D graphics. When we started to use the Tk canvas with a view towards implementing various graphical diagram editors, however, we realized that we needed a more powerful layer of abstraction.

To illustrate, figure 1 shows a mock-up visual program of the kind that we were interested in implementing. This is a sample of a language developed in [10]. The rectangle marked "let" encloses an expression, and the value of the expression is indicated by the arrow connecting the two triangular "terminals." The let-expression itself is connected to another function box. Items within the let-expression can be moved only within that frame, while moving the whole frame will move everything contained within it. This combination of hierarchy and complex user interaction necessitated a higher-level framework than the raw canvas.

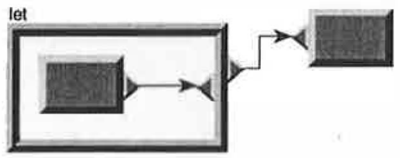


Figure 1: A fragment of a visual program

We decided very early on that our implementation would be only in [incr Tcl], in order to guarantee portability. More recently, we used the namespace facility of Tcl 8.0 to port the code to Tcl 8.0, and it now runs in either Tcl and [incr Tcl]. Although the Tcl-only decision presented its own set of implementation challenges, we were able to achieve acceptable performance and the portability we desired. The Slate should, in fact, work with *any* canvas extension, since it uses only the standard canvas interface provided by Tcl/Tk.

After some experimentation, we ended up with a package that implements the following:

User-defined items The key feature that makes the Slate useful is the ability to define new item types in Tcl. Items can be composed recursively, producing a straight-forward visual hierarchy, as is common in many graphics packages (see, for example, [2]). All of the canvas methods are rewritten to handle hierarchical items.

Item shapes Every item has a shape, such as *point*, *rectangle*, *polygon*, or a custom-designed shape. Items can be queried for the coordinates of a *feature*, such as the north-east corner or the second vertex. Items can be requested to move one or more features, reshaping the item.

Interactors Event-handlers can be bound to any level of the visual hierarchy. In addition, we implemented a more abstract and more powerful user interaction framework, in which par-

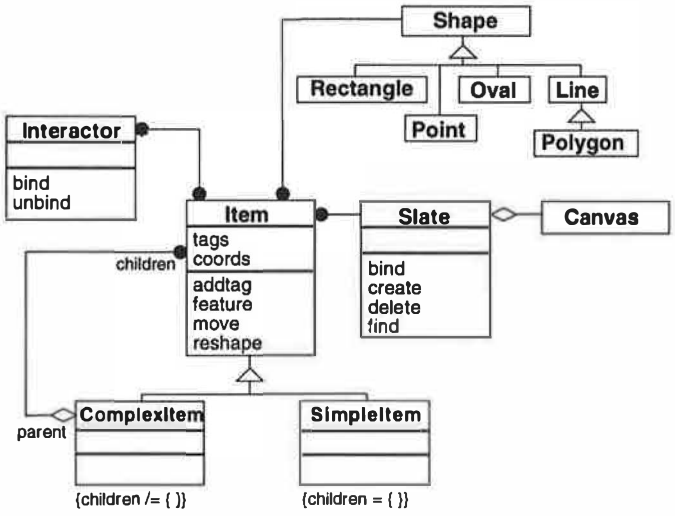


Figure 2: The key framework classes

ticular sequences of user interaction events are abstracted into objects called *interactors*.

We also added other useful methods, such as highlighting and selection. Unlike many research toolkits, we did not implement a constraint system, for reasons detailed in a later section.

2 The core graphics framework

The Slate is implemented as a set of classes, organized around one key class called *Slate*. We have tried to make the Slate a plug-in replacement for the Tk canvas. A simple example:

```
set slate [::tycho::slate .s \
      -height 300 -width 400 \
      -background white]
pack $slate -fill both -expand 1
```

Figure 2 shows a conceptual representation, in the Object Modeling Notation [11], of the key classes. The diagram is conceptual only, because many of these classes do not in fact exist – in the implementation, they are faked using Tcl procedures, canvas item tags, and associative arrays.

The *Slate* class is wrapped around a Tk canvas. It contains an arbitrary number of *Items*, which are graphical elements that appear on the screen. An Item is either a *ComplexItem*, which is in turn an aggregation of Items, or a *SimpleItem*, which represents a single Tk canvas item such as a line or rectangle. Each item has a *Shape* (see section 2.3), and can be operated on by an arbitrary number of *Interactors* (see section 3.2).

2.1 Complex item classes

To add a new item type to the slate, the programmer subclasses the *ComplexItem* class. Once defined, items of the new type can be created and manipulated just like regular Tk canvas items. For example, one of the item types that we supply with the Slate is called *Frame*, because it mimics the appearance of the Tk *frame* widget. To create a new frame on a slate, we can execute code such as this:

```
set frame [$slate create Frame \
        50 50 100 100 \
        -color green -relief ridge]
```

which produces the item shown at the top left of figure 3. This item behaves like any other Tk canvas item – for example, we can change its coordinates:

```
$slate coords $frame 70 70 140 120
```

We can move it:

```
$slate move $frame 40 20
```

We can get a list of items overlapping a given region of the canvas, which will (in this case) include this item:

```
set found [$slate find \
      overlapping 100 100 200 200]
```

Figure 3 shows several other complex item types. At the top right is a *Solid*, which is a polygon with a pseudo-3D border like *Frame*; at the bottom left is a *LabeledRectangle*, which is a rectangle with a label and arbitrary graphics nested within it (in this

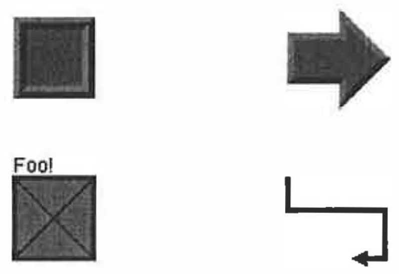


Figure 3: Some pre-defined complex items

case, two lines); at the bottom right is a *SmartLine*, which is a line that, given two end-points and the directions at those ends (n, s, e, or w), draws itself as one or more orthogonal segments. We emphasize that these items are a sample of those that we found useful for building graphical editors, and that it is quite simple to create new item types or to extend existing types by subclassing.

2.2 Constructing hierarchy

Any subclass of *ComplexItem* can add items to itself, thus creating a recursive hierarchy of items. For example, a *Frame* item consists of four simple items: a rectangle for the central surface, two polygons for the "lit" and "shaded" borders, and a transparent rectangle that is used as a place-holder for the coordinates of the whole *Frame*.

In addition to creating hierarchy by creating new item types, an instance of the *ComplexItem* class can have arbitrary sub-items added to it. To illustrate, we can create a blank complex item:

```
set citem [$slate create ComplexItem \
          50 50 100 100]
```

The coordinates give the region to be occupied by the item so that methods such as *coords* will operate correctly. Now we can add items to it. For the sake of example, let's add a pair of lines and an oval to it:

```
$slate createchild $citem line \
          50 50 100 100
$slate createchild $citem line \
          50 100 100 50
$slate createchild $citem oval \
          60 60 90 90 -fill green
```

The item that results is shown at the left of figure 4. Like any slate item, this item responds to methods that move and scale it. For example,

```
$slate coords $citem
```

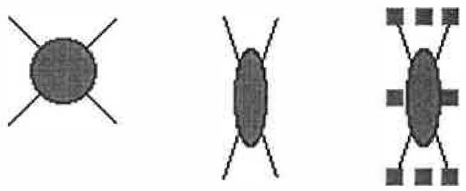


Figure 4: A dynamically-constructed complex item

will return {50 50 100 100}. The code

```
$slate scale $citem 50 50 0.5 1.5
```

will scale the item, producing the item at the center of figure 4.

In general, arbitrarily complex items can be built up this way. Figure 5 shows one such hierarchy, similar to those we use in one of our graphical editors. As a general rule, a programmer should define a new subclass of *ComplexItem* when a particular graphical representation is used again and again, and use dynamic composition of items, such as just given, when items are combined as part of the editing operations in a graphical editor.

2.3 Shape

Each item on a slate has a *shape*. Shapes provide more sophisticated control over the coordinates of an item than just its raw coordinates. Simple items have a shape that cannot be changed; complex items have a shape that is determined by the class defining that item. Predefined shapes mimic the primitive canvas item types: *point*, *rectangle*, *oval*, *line*, and *polygon*.

An item with a given shape has a set of attributes called *features*. Features are inspired by Gleicher's work on constraint-based graphics [3]. A feature is typically a point location on the item. An item can be queried to find the value of a feature, and the feature can be moved to change the shape of the item. For rectangular items, the default features are its center, the four corners, and the four edges; for lines, the features are the vertices of the line.

For example, to find the coordinates of the northwest corner of a rectangular item, we could execute:

```
set northwest [$slate feature $frame nw]
```

To reshape the item by moving the north-west corner left and down ten pixels, we could execute:

```
$slate reshape $frame -10 10 nw
```

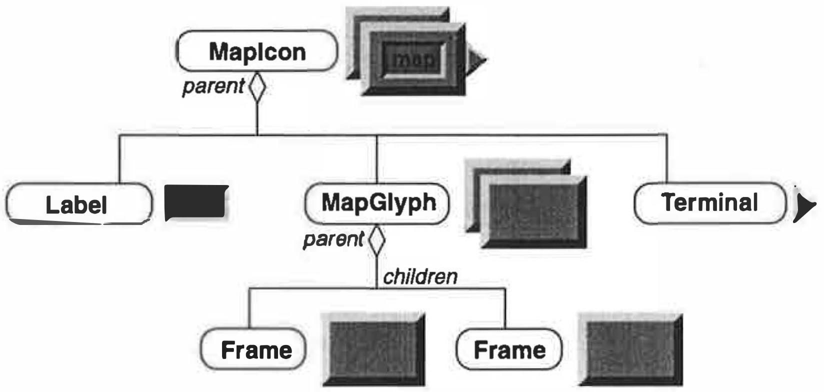


Figure 5: A sample visual hierarchy

All features can be read, but not all can be set; *center*, for example, can be read but not set. Any feature that can be set can also be *grappled* – that is, have a grab handle attached to it. For example, executing the code:

```
$slate grapple $citem
```

will add grab handles to the four corners and edges of the given item. The effect of executing this code is shown on the right of figure 4 – this item can be resized by dragging any of the grab handles.

The *Shape* classes are implemented as collections of class procedures – each class handles feature queries and reshaping of all items with that shape. Complex items can choose their own set of features by overriding their shape-related methods. For example, the *Terminal* item in figure 5 has features called "origin" and "terminal," which are its base and connection point respectively.

2.4 Tags

Tagging in the Slate is a fairly straight-forward extension of the way that tags are implemented on the Tk canvas. Any item can be tagged. For example, given some item *citem*, we can write

```
$slate addtag "fred" withtag $citem
```

Some time later, we could write

```
$slate move "fred" -10 0
```

which would move *citem*, including all of its components, ten pixels to the left. (Any other item tagged with "fred" will also be moved.) Any node in the hierarchy can be tagged in this way, and performing an operation on the tag will operate on the corresponding subtree of the hierarchy.

3 Interaction mechanisms

The visual hierarchy provides an elegant means of constructing drawings, but does not by itself provide user interaction. To effectively support construction of visual language editors, we need to provide ways of adding user interaction.

3.1 Bindings

The first interaction mechanism supported by the slate directly mimics the binding mechanism of the Tk canvas. With this mechanism, a command can be bound to an event and an item. For example, executing

```
$slate bind $citem <Button-1> {puts !!}
```

will add a binding to *citem*. Whenever the mouse is clicked on item *citem*, the string "!!" is printed to the console.

In the presence of a visual hierarchy, bindings take on some subtle complications. In figure 5, for example, I want a click on either frame to be handled by the top-level item. Dragging these items should move the whole tree of items. The terminal item, however, should respond differently – in this case, I want clicking and dragging on the terminal item to create a new arrowed line and extend the end of the line to follow the cursor. Also, dragging on the text label will sometimes need to select a region of the text.

Our solution is to *mark* nodes of the tree: only marked nodes are able to respond to user input. Figure 6 illustrates a marked tree, in which nodes *a* and *e* are marked. With respect to any node, its *root node* is the root of the lowest marked sub-tree containing it. The top-level node is the root of the whole tree, and is implicitly marked. Conceptually,

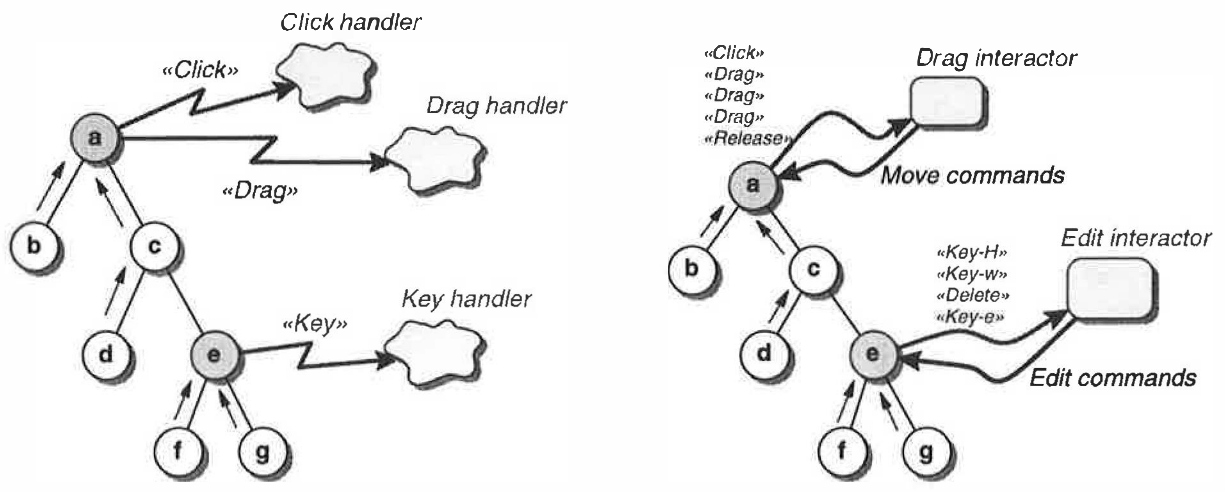


Figure 6: Events and interactors in the visual hierarchy

when an event occurs on an item, the event is propagated up the tree until it reaches a marked node, at which point the event is handled. This is illustrated at the left of figure 6: node *a* will handle events from itself, *b*, *c*, and *d*; node *e* will handle events from itself, *f*, and *g*.

In the current implementation of the Slate, nodes can only be marked when they are created. Returning to our earlier example, we can create a child item of a *ComplexItem* that responds to user input with code such as this:

```
set flag [$slate createrootchild \
      $citem rectangle \
      40 50 50 60 -fill red]
$slate bind $flag <Button-1> \
      {puts Foo}
```

Events can also be used with tags. For example, we can write

```
$slate bind "fred" <Button-1> \
         {puts "Clicked fred!"}
```

and any top-level or marked child item tagged with *fred* – that is, it and the nodes for which it handles events – will respond to the event.

If used without any hierarchy, the slate thus provides the same mechanism as the canvas for event-handling. The visual hierarchy aids more complex item construction without discarding this powerful mechanism. However, the binding mechanism is low-level, and complex user interaction built on this mechanism very quickly mushrooms into spaghetti-like code.

3.2 Interactors

The second interaction mechanism is based on *interactors*, proposed by Myers in 1990 [7] and implemented in the Garnet toolkit and its successor, Amulet [8, 9]. Interactors abstract user interaction from the lower-level events upon which they are built, and in the process modularize the code and make it more re-usable.

An interactor is an object that intercepts events and translates them into operations on a *target* item. For example, a *Follower* interactor – so called because it "follows" the mouse – translates mouse events into calls to the *moveclick*, *movedrag*, and *moverelease* methods of the slate. To create a Follower interactor, we can execute, say:

```
set follower \
    [$slate interactor Follower]
```

To make an interactor operate on an item, we *bind* the interactor to that item. For example,

```
$follower bind $frame -button 1
```

Now, dragging the frame item with the mouse makes it move – simple! To stop the item from responding to the mouse, unbind the interactor:

```
$follower unbind $frame -button 1
```

The right side of figure 6 illustrates two interactors bound to nodes of a hierarchy. Interactors can be cascaded to create more complex interaction. For example, an interactor that moves an object only within a certain region of the slate can be cascaded with an interactor that quantizes movement to ten-pixel steps.

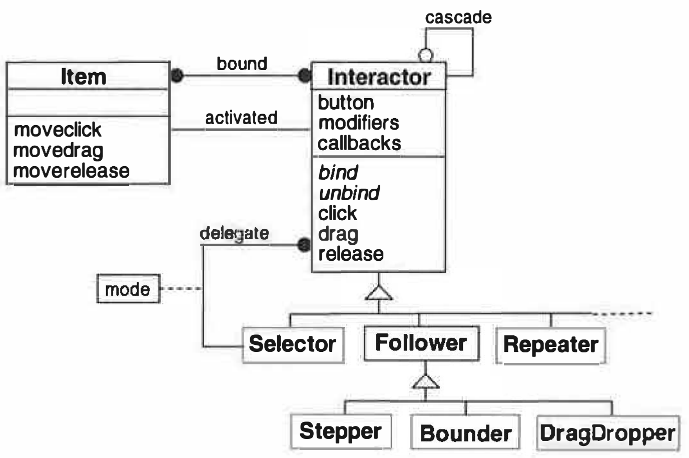


Figure 7: The interactor classes

A more complex combination is achieved with the *Selector* interactor, which manages a graphical selection in the same manner as typical drawing programs. When items become selected, the Selector *delegates* interaction events to another interactor according to various conditions that can be set up by the programmer. For example, it might forward mouse and keyboard events to a *LineEditor* interactor if there is only a single text item selected, and forward mouse events to a *Follower* interactor if more than one item is selected.

Interactors allow user interaction code to be modularized, and, just as importantly, reused. In addition, they allow highly dynamic user interaction, such as changing the effect of the mouse depending on the context (such as the number of items selected). This kind of dynamic modification of user interaction would be very difficult if coded directly using event bindings. Some additional examples of interactors are given in section 5.

4 Implementation notes

As mentioned previously, the ComplexItem "class" and its subclasses, which define new item types, are not really implemented using classes. Because the natural implementation of complex visual elements in Tk utilizes item tags to avoid recursive tree-walks, we flattened the object structure as well. (In a language that supports fine-grained objects, such as Java, we would use separate objects and recursive tree-walk algorithms.)

Each "class" is thus a collection of procedures that accept a slate, the contained canvas, and an item ID as the first three arguments. Methods of the slate call procedures in the appropriate namespace when necessary – for example, a command such as

```
$slate create Frame 40 40 80 80
```

will call the *construct* procedure in the Frame namespace. To simulate inheritance, each namespace contains an associative array mapping method names to the appropriate procedure, which is overwritten in each "sub-class."

We used some other tricks as well. For example, to access option variables and instance variables, we index a shared array by a combination of a name and the ID of the item. This allows the Slate uniform access to the internals of complex items, regardless of class. For example, a procedure in Frame that accesses its *color* option would use:

```
set foo $option(color$id)
```

We use tags to simulate hierarchical complex items. Although this seems conceptually simple, actually doing it in the presence of event bindings is a little tricky. Each complex item is assigned a unique ID when it is created – we use an integer preceded by an underscore (the Tk canvas uses plain integers). Every simple item within a complex item is tagged with that ID:

> *Tagging rule 1*: Every simple item within a complex item is tagged with that item's ID.

The corollary is that every simple item is tagged with the IDs of all items above it in the hierarchy – this makes moving a subtree easy. Now, user interaction often requires finding the root item containing a simple item. To make this efficient, each simple item within a hierarchy is given a special tag:

> *Tagging rule 2*: Every simple item contained in a complex item is tagged with the ID of its root prefixed by "!".

The corollary of this rule is that any simple item that does not have a tag starting with "!" is not in a hierarchy. With these rules, it is easy (conceptually – the implementation is a little tricky) to find, move, and manipulate complex and simple items.

Now, some additional rules are needed to effectively deal with event bindings (and, by extension, interactors). Since a simple item can have only one root, it can have *at most* one tag beginning with "!". Therefore, when binding an event to a complex item, we follow this rule:

> *Binding rule 1*: To bind an event to a complex item, bind to the tag constructed by prefixing its ID with "!".

The net effect is exactly as it would be if events were propagated up the hierarchy until a marked node were found. Finally, events can also be bound to tags, which is the same as binding to a tag on the canvas:

> *Binding rule 2*: To bind an event to a tag, bind the event to that tag on the canvas.

The implementation of most slate methods based on these rules is reasonably straight-forward. Each method tests for three types of argument – a simple item ID, a complex item ID, or a tag – and acts accordingly. For example, a typical method has the pattern:

```
if { [string match {[0-9]*} $tag] } {
    # Process a canvas item
    ...
} elseif { [string match {_*} $tag] } {
    # Process a complex item
    ...
} else {
    # It's a tag: find matching items
    set items [find withtag $tag]
    ...
}
```

5 Building useful tools

The Slate itself is only part of the infrastructure needed to build useful tools. In this section we briefly describe our implementation of some tools that use the Slate.

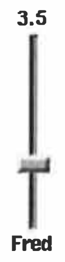


Figure 8: A custom slider widget

5.1 Custom widgets

One of our first uses of the Slate was to build a custom slider widget, shown in figure 8. (It looks much better in real life when you have several lined up together.) This slider widget mimics the sliders used in audio control equipment in appearance, but mimics the Tk *scale* widget in behavior. Four interactors are used to produce the desired user interaction.

Figure 9 shows code for a simplified version of this widget. The four sections of this code:

1. Create the four items show in the diagram: two text labels and two pseudo-3D rectangles.

2. Create and bind a *Bounder* interactor, which moves the slider bar up and down and keeps it within the desired limits.

3. Create and cascade a *Stepper* interactor, which quantizes movement to multiples of 0.5 (in this example).

4. Define a procedure that is called whenever the bar is moved. The procedure calculates the value represented by the bar, and updates the numeric text item.

In the real Slider widget, there are two other interactors that i) step the slider towards the mouse if the left button is clicked on the background, and ii) cause the bar to jump to the position of a button-2 click and then follow the cursor.

All told, constructing this widget was relatively easy, and we didn't have to write a single event binding.

5.2 Graphical editors

Figure 10 shows a snapshot of one of the graphical editors constructed using the Slate. This editor is the front-end for Ptolemy II, a new version

```
# Create the display elements
set value [$slate create text 50 20 -text 0 -anchor s -fill blue]
set trough [$slate create Frame 48 23 52 143 -color darkgrey \
        -borderwidth 2 -relief sunken]
set bar [$slate create Frame 40 132 60 142 \
        -color darkseagreen -borderwidth 3]
set label [$slate create text 50 150 -text "Fred" \
                -anchor n -justify center]

# The bounder moves the bar along the trough
set bounder [$slate interactor Bounder \
        -dragcommand "updateWidget $slate $bar $value" \
        -constrain y -bounds {0 24 0 142}]
$bounder bind $bar -button 1

# The stepper quantizes movement to increments of 0.5
set stepper [$slate interactor Stepper -stepsize [expr 108.0/22]]
$bounder cascade $stepper

proc updateWidget {slate bar value args} {
    set position [expr [lindex [$slate coords $bar] 1] + 5]
    set x [expr (137.0-$position)/108.0 * 10.0]
    $slate itemconfigure $value -text [format %.1f $x]
}
```

Figure 9: A simple example of a custom widget

of Ptolemy [6] being written in Java. The particular system shown is a second-order continuous-time simulation, written by Jie Liu of UC Berkeley. In this editor, icons can be selected from a library stored as ASCII text, which describes each icon in terms of the ComplexItem type used to draw it, the number and location of terminals, the label, and the graphics to draw upon the surface of the icon.

Once placed, icons can be selected and moved around. (This is done using the Selector and Follower interactors.) When the mouse moves over an unconnected terminal of an icon, a *DragDropper* interactor is activated, which highlights the terminal to indicate that it is "ready." If the mouse is clicked on the terminal, the DragDropper creates a new SmartLine item, and reshapes the line so that the end follows the cursor. When the end of the line moves over a terminal, the DragDropper activates a call-back to test if the terminal is a suitable "drop target." If it is, it snaps [4] the end of the line to the connection point of the terminal, altering the shape of the line to make it join at the expected angle.

Internally, this graphical editor uses a variant of the model-view-controller architecture (see, for example, [1]). As the user places icons and connects terminals, the interactors forward events to either an edge controller or a vertex controller. The architecture is shown in figure 11. The controller first decides what the user interaction means in terms of the underlying semantics of the visual program, and modifies the *semantic model* accordingly (in this example, the semantic model is a directed graph). The controller also decides what visual aspects of the program have changed, such as moving the end of a line, or adding a new icon. It modifies the *layout model* accordingly, which in turn notifies the *view* containing the slate, which in turn updates to reflect the new appearance.

This seems, at first, somewhat complicated, but our experience indicates that it does leads to highly modular and customizable editors. We are, however, still gaining experience with this architecture. Interactive response is very good, partly because the interactor model is able to optimize incremental mouse movements while items are being dragged about on the screen.

Finally, we note here our observations on the use of *constraints*, as included in many experimental toolkits ([8, 9], for example). In constraint systems, the programmer sets up constraints between graph-

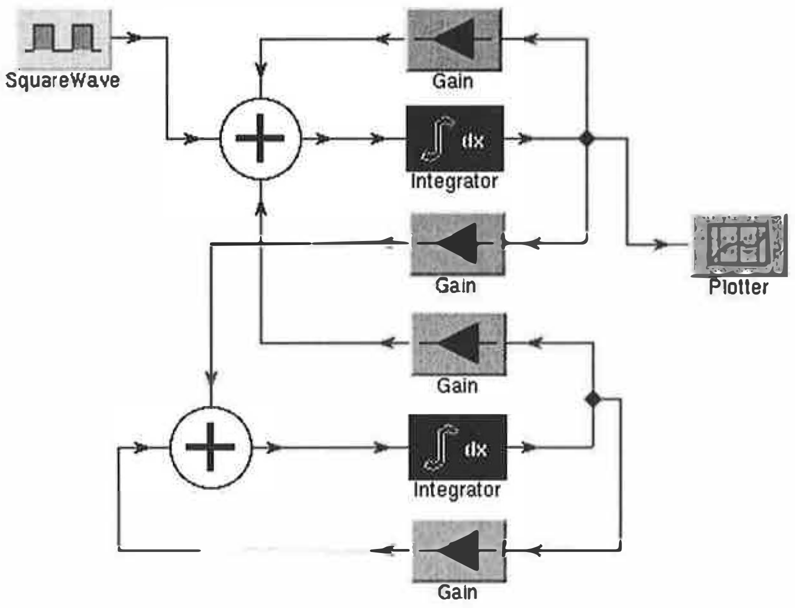


Figure 10: A snapshot of a graphical editor

ical items, which the constraint system attempts to maintain as conditions change. For example, our editor could have constraints set up such that the ends of attached lines move when an icon is moved.

We wrote a simple constraint system early on, and found that writing constraints to catch the right conditions and to avoid cycles was tricky. We ended up trashing the constraint system – simple though it was – when we realized that the model-view-controller architecture offers a better and simpler alternative. If you look at figure 11, you can see that, as an icon is moved, the layout model must be modified for attached edges as well as the icon. How do we know what edges to move? Why, by looking at the semantic model!

Thus, we concluded that, in interfaces that have an underlying semantic model, using that model with an appropriate controller object is a more straight-forward solution than general-purpose constraint solvers. In addition, since we have a comprehensive set of interactors, it is also straight-forward to implement purely-syntactic constraints by appropriately configuring interactors. For example, keeping the slider in figure 8 on the right track does not require constraints, just the right interactor.

6 Concluding remarks

The Slate is, we feel, a powerful and useful tool that provides a significant increase in abstraction over the Tk canvas. Because it is written entirely in Tcl/[incr Tcl], it is highly portable and should work with any other canvas extension.

The Slate is part of the Tycho user interface system [5], which can be obtained from the Tycho home page:

```
http://ptolemy.eecs.berkeley.edu/tycho/
```

Current development versions of the Slate can be obtained from:

```
http://ptolemy.eecs.berkeley.edu/
              ~johnr/code/slate
```

Acknowledgments

Key infrastructure without which this project would not have been possible has been developed by: Michael McLennan of Bell Labs ([incr Tcl]/[incr Tk]), John Ousterhout of Scriptics Corporation (Tcl/Tk), and Mark L. Ulferts of DSC Communications Corp ([incr Widgets]). We also thank the reviewers for their careful reading and comments on this paper.

Tycho is part of the Ptolemy project, which is supported by the Defense Advanced Research Projects Agency (DARPA), the State of California MICRO program, and the following companies: The Alta Group of Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, NEC, Philips, and Rockwell.

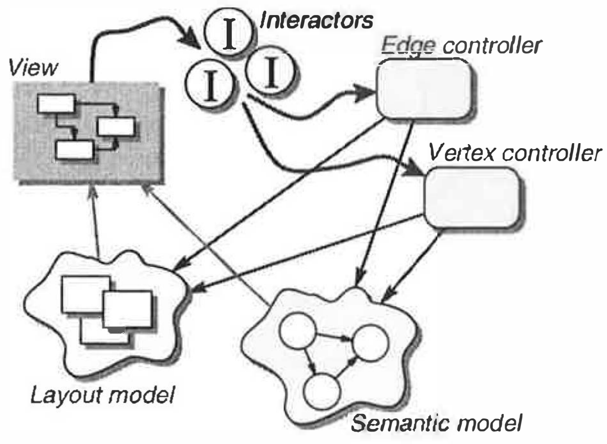


Figure 11: The architecture of a graphical editor

References

[1] Dave Collins. *Designing Object-Oriented User Interfaces.* Benjamin/Cummings, 1995.

[2] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wedley, 1996. Second Edition.

[3] Michael Gleicher. *A Differential Approach to Graphical Manipulation.* PhD thesis, Carnegie Mellon University, 1994. Also appears as CMU School of Computer Science Technical Report CMU-CS-94-217.

[4] Scott E. Hudson. Semantic snapping: A technique for semantic feedback at the lexical level. In *Proc 1990 SIGCHI Conference*, pages 65–70, April 1990.

[5] Christopher Hylands, Edward A. Lee, and H. John Reekie. The Tycho user interface system. In *The 5th Annual Tcl/Tk Workshop '97*, pages 149–157, 1997.

[6] Edward A. Lee and David G. Messerschmitt et al. An overview of the Ptolemy project. `http://ptolemy.eecs.berkeley.edu/papers/overview/`, March 1994.

[7] Brad. A Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, July 1990.

[8] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(1), November 1990.

[9] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doan. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

[10] H. John Reekie. *Realtime Signal Processing: Dataflow, Visual, and Functional Programming.* PhD thesis, School of Electrical Engineering, University of Technology, Sydney, Australia, September 1995.

[11] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

Iclient/Iserver: Distributed Objects using [incr Tcl]

Lee F. Bernhard
Bell Labs Innovations for Lucent Technologies
Now at Scriptics Corporation
lfb@scriptics.com

Iclient/Iserver is a simple distributed object framework for [incr Tcl] applications that enables its clients to synchronize activities and share information. Using Iclient/Iserver, clients can access objects living on a remote server tranparently, making building client/server applications both easy and intuitive. Iclient/Iserver is conceptually similar to the widely used CORBA standard, but is much simpler, intended for building smaller, client/server applications where the cost and complexity of a CORBA implementation cannot be justified

In this paper I describe the use of Iclient/Iserver for sharing server objects among clients. I explain the underlying architecture and implementation of the distributed object system. I conclude by illustrating how to use Iclient/Iserver to build a simple networked version of the card game Hearts.

Introduction

Since the Tcl/Tk core added sockets and safe interpreters, using Tcl/Tk to create networked applications is very simple. This simplicity opens up a realm of applications in which users can communicate and share information using Tcl/Tk over a network. With the Tcl/Tk Plugin[1], programmers can bring a networked application right to the user's desktop via a web browser.

Networked applications are usually developed in the client/server paradigm, in which a server provides central services, and client programs access these services over a network. A server might control a database, and clients might communicate with the server to query and manipulate the data. Or a server might host a multiplayer card game, and clients might connect in to play the game.

In all of these applications, the server centralizes shared data. It synchronizes the actions of the clients, so that, for example, one client can't access a data record while another is updating it. The server also broadcasts significant changes, so that when one client modifies a data record, other interested clients automatically know about it.

A significant problem in writing client/server applications lies in defining the application programming interface (API) between the server and the client. The server has some subset of procedures that it wants to publish to the client to allow the client access to its services; the difficulty lies in providing an API that publishes these procedures in a way that is both simple and manageable in terms of size and complexity. In a networked card game, there might be a procedure to query the list of games, a procedure to join a game, a procedure to query a hand, a procedure to play a card, a procedure to check the score, and so on. Even for a simple application like Hearts, the number of access procedures can quickly get out of hand.

Iclient/Iserver greatly simplifies the development of client/server applications by allowing the programmer to implement clients and servers with Tcl/Tk and the [incr Tcl] object system[2][3]. With just a few lines of code, programmers can set up an object server that publishes its objects to any connected client(s). In a stand-alone [incr Tcl] script, class methods provide the major interface for performing operations and manipulating data. By publishing its objects, the server allows clients to use the same interface that the server script uses to access its objects. The server's existing object system becomes the API for clients to access server services, eliminating the bother of writing procedural wrappers around each server object.

Iclient/Iserver is not the only solution to provide distributed objects; perhaps the most widely recognized solution to providing distributed objects is the Common Object Request Broker Architecture (CORBA)[4][5], a distributed object standard designed by the Object Management Group (OMG) after eight years of discussion and collaboration. It uses a model of an "object bus" that clients use to access remote objects. CORBA builds upon the successes and services of previous technologies like RPC and TP monitors. It is an ambitious framework, allowing clients to access objects implemented in a different language than the client program. But its ambition also makes it a complicated framework to learn and work with; the complexity is hard to justify when creating smaller client/server applications. Additionally, CORBA's goal of allowing multiple languages to share components requires a translation layer (IDL) that isn't necessary, for example, in building small Tcl/Tk applications.

There is already support for procedural client/server development in the Tcl/Tk community. The Tcl-DP extension [6][7] makes it easy to set up RPC-style client/server applications with a procedural API. It includes some support for simple, slot-based structures. The GroupKit extension [8][9] is another RPC-based solution with slot-based structures, designed with collaborative applications in mind.

There are a number of extensions that allow Tcl scripts to access CORBA objects and services. These extensions make it easy to write Tcl scripts that control CORBA distributed components, and several have facilities for creating Tcl structures that other CORBA clients can access. They don't work seamlessly with [incr Tcl]; they require class descriptions written in a neutral language format, IDL.

Iclient/Iserver is similar to these packages in some respects, occupying a space somewhere between the convenience and simplicity of Tcl-DP and Groupkit and the robust object support of CORBA. It lets both clients and servers take full advantage of object-oriented technology. Instead of using a simple, slot-based object model, Iclient/Iserver uses [incr Tcl] to provide a full-featured, class-based model. Classes can encapsulate data and related operation, with support for public, protected, and private members. [incr Tcl] also supports single and multiple inheritance, so that classes can share functionality.

Iclient/Iserver harnesses [incr Tcl]'s existing code base and adds a quick, easy way to distribute objects. With Iclient/Iserver, programmers can concentrate on designing classes that provide core functionality; they can add the client/server capability almost as an afterthought.

Counter: A Simple Example

Suppose you have a `Counter` class with an object named `foo`. This object contains a number, which programmers can increment and query using the methods `bump` and `value`. An object like this could help clients to generate serial numbers for processes, orders, transactions, etc. Programmers can create the object on the server and make it available to clients with the following code:

```
package require Iserver
class Counter {
    inherit ::iserver::Distributed
    method bump {} {return [incr count]}
    method value {} {return $count}
    private variable count 0
}
Counter foo
iserver::listen 8066
```

The server begins by initializing the `Iserver` package, thereby creating commands and base classes that the server needs. It then defines an [incr Tcl] class `Counter` with two methods `bump` and `value`. By inheriting from the `Distributed` class, the `Counter` class is able to share its objects with clients. The server creates a `Counter` object called `foo`, and then opens a socket on port `8066` and waits for clients to connect and start using `foo`.

Now suppose there are two clients that would like to use the server's `foo` object. Each client needs to load the `Iclient` package, and then connect to the server. It does so by creating a `Server` object, and telling it to connect to the server listening at port `8066` on the machine `sercial.micro.lucent.com`.

```
package require Iclient
set serv [::iclient::Server #auto \
  sercial.micro.lucent.com 8066]
$serv resolve class Counter
$serv resolve object foo
foo bump
```

Now that the client is connected to the object server, it can attach itself to the `Counter` `foo`. To do this, the client resolves the names of both the class, `Counter`, and the object, `foo`. In other words, the client has gone to the server and created its own, local copy of `Counter` and `foo`, which it can use to access the actual, server entities of the same names. All the client needs to do is call the `bump` method on its object, and Iclient will take care of running `bump` on the server object `foo`.

The local copy of `foo` is merely a client-side stub. Methods are not truly implemented on the client side; instead, the client methods serve as an interface for calling corresponding methods on the server object `foo`. A client stub can also be considered conceptually as a local, client-based, reference to a server object.

Now imagine that the clients want to use the `Counter` `foo` in a simple GUI. A label will store the current value of `foo`, and a button will allow the client to `bump` the value of `foo`. If one client changes the value of `foo`, the GUI of the other client should update to show the new value as well.

```
...
button .bump -text "Bump" -command {
  foo lock {
    .count configure -text [foo bump]
  }
}
pack .bump -side left -padx 6 -pady 10
label .count -text [foo value]
pack .count -side left -padx 6 -pady 10

foo watch lock {
  .count configure -text [foo value]
}
```

This example uses two object services of Iclient/Iserver: `lock` and `watch`.

When pressed, the `.bump` requests a lock on `foo`. It then increments its value by calling the `bump` method.

Interprocess communication presents a host of concurrency problems that occur when two clients try to manipulate the same resource at the same time. In Iclient/Iserver, most method calls will be atomic, unless they access the event loop. To be certain of exclusive access to a server object, a client should request a lock whenever modifying a server object.

Locks are especially helpful when a client needs to manipulate a server object with multiple method calls, where the entire series of operations needs to run atomically. By acquiring a lock on a server object, a client gains exclusive control of that object.

In a slightly more complicated scenario, clients may be using the `Counter` object to keep track of the current bid price in an auction. Clients will check the current price, and if it falls below `maxBid`, they will bid by incrementing the `Counter foo`.

```
if {[foo value] < $maxBid} {
  foo bump
}
```

What makes this example different is that the clients want to check the state of an object, make a decision based on that state, and then call another method on the object to change it. Without locks, there is a classic race condition where it is possible for a client to make the wrong decision about whether to bid.

Imagine that the bid price is one less than `maxBid` for each client. The object server processes client requests in the order they are received. If both clients check the value of `foo`, they will both receive the same value and decide to `bump` the `Counter`. When this occurs, one of the clients will exceed their maximum bid, because the state of `foo` will have changed between the time the client checked its value, and the time it tries to `bump` the `foo`.

Locks solve this problem nicely by allowing each client to check `foo` and `bump` its value without interruption from the other client. immediately:

```
foo lock {
  if {[foo value] < $maxBid} {
    foo bump
  }
}
```

This is guaranteed because once the first client acquires the lock on `foo`, it is the only client allowed to access `foo` until the lock is removed. If another client tries to access `foo` while the first holds the lock, it will have to wait until the lock is released.

By default, clients will ten seconds for the lock; if that time expires, the `lock` method raises a Tcl error. Clients can adjust the timeout factor by providing a value, in milliseconds, for the `-timeout` option of the `lock` method. A `timeout` of zero indicates that the client should give up waiting if it cannot obtain the lock. If the client may need to wait for a noticeable period to obtain a lock, it will likely wish to display a watch cursor, or some other GUI indication that the client is busy. The `lock` method provides two more options, `-suspend` and `-continue` to allow the client to run a script immediately before waiting for the lock, and immediately after the lock is obtained. The `Counter` clients may use this to display a message in the label `.count` as they wait for the lock:

```
foo lock {
  .count configure -text [foo bump]
} -timeout 20000 -suspend {
  .count configure -text "Acquiring lock"
} -continue {
  .count configure -text "Lock acquired"
}
```

One more detail remains to be explained: when one client bumps the value of `foo`, the other clients need to know that `foo` has changed in order to update their displays. Clients can `watch` objects, and register a callback that will run when a particular event occurs to the object. This is similar to a variable trace or to a binding on a Tk widget. Keeping track of an object being locked is rather straightforward. The clients can `watch` for `foo` to be locked; when that occurs, they know that `foo` has changed, and they can call the `value` method to learn the new value of the `Counter`. Clients can `watch` any object, looking for well-defined or custom-defined events to occur. Well-known events include `lock`, `destroy`, and `resolve`. `Distributed` objects can report custom events by using the `report` method, as in the following:

```
class Counter {
  inherit ::iserver::Distributed
  method bump {} {
    report "bump"
    return [incr count]
  }
  ...
}
```

The last services that Iclient/Iserver provide are the ability to restrict use of a server object to a subset of connected clients. An object server gives each of its clients an unique id, and can use this id with the `Distributed` class's `restrict` method. Here a server restricts use of the object foo to a single client with client id `client1`:

```
foo restrict {client1}
```

Architecture

Applications written using Iclient/Iserver are constructed with a client/server architecture where the server program provides central services for the client programs connected by a network. The server helps the clients to interact by holding shared information and synchronizing the clients' access to this information.

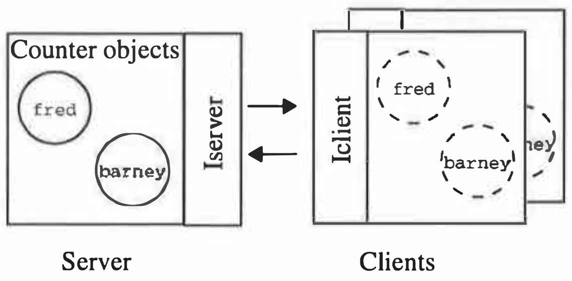
In any client/server architecture, an API mediates between clients and server services. In the remote procedure call (RPC) approach, the server program publishes a set of commands that clients use to communicate with the server. An RPC approach works well for procedural applications, such as vanilla Tcl scripts, where command provide the only interface for performing operations.

An [incr Tcl] script uses classes to encapsulate data and related commands into a single entity. Each class has a well-defined set of methods that act as the interface for working with objects of that class. By invoking these methods, it is possible to access and manipulate the data contained by the object.

Turning an [incr Tcl] script into a client/server application should be a process that keeps the interfaces defined by the classes intact. A RPC approach does not work well with [incr Tcl] scripts because it requires wrapping each method with a procedure. there to be a procedural wrapper for accessing the methods of an object. Adding a procedural communication mechanism to an object-oriented design breaks up the design of classes. At best, it is inelegant. At worst, it is tedious to set up and confuses the interface created by the class methods. It would be much better to allow clients to access server objects in the manner that the server manipulates its object: by calling the methods defined for the object.

That is the goal of Iclient/Iserver. The server uses [incr Tcl] to create classes and objects with well-defined interfaces. Iserver provides a base class, `Distributed`, that publishes derived classes and their objects, so that clients connecting to the server can use those objects across a network socket.

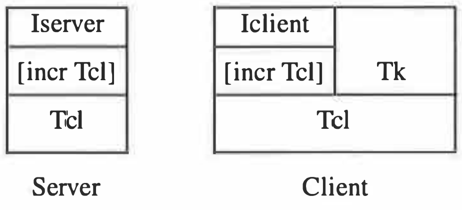
Iclient provides the client access to server objects by creating local copies, or stubs, which act as references to the server objects. When a client access a client stub, the stub encapsulates the server communication required to access the corresponding server object.



Server objects are visible on Clients

It is possible for a client to be connected simultaneously with multiple object servers. Iclient provides a class `Server` to help client scripts to keep track of a given object server connection. When a client wants to connect to a new object server, it creates a new instance of this `Server` class and feeds it information about the hostname and port number of the object server. The methods of the `Server` class provide a secondary API that clients use to access an object server, to create client stubs, create new server objects, destroy server objects, and inquire about objects available on the server.

Tcl Package Architecture of Client and Server



Iclient and Iserver are currently implemented as Tcl scripts that rely upon [incr Tcl] to provide classes and objects. The first alpha of Iclient/Iserver used [incr Tcl]'s namespace facility, but the revised version uses [incr Tcl] 3.0 and Tcl 8.0 namespaces. Iclient and Iserver use namespaces to encapsulate the classes and commands that they introduce.

Distributed Object Model

This section describes in detail how the Iclient/Iserver distributed object model works. It covers the two ways clients can create client stubs for server objects. It discusses the underlying commands and classes that the stubs use to communicate with the server. It also describes how sockets, fileevents, and safe interpreters form the primitive communication framework.

The central task for Iclient/Iserver is to allow clients to access server objects in a manner consistent with accessing ordinary [incr Tcl] objects in a stand-alone application. To do this, a server object must appear as if it lived locally, on the client. It must also have the same methods as the server object, and the client must be able to call those methods as with ordinary [incr Tcl] objects.

To accomplish this, Iclient/Iserver literally creates a client-side object to act as a stub for the server object the client would like to use. These stubs are [incr Tcl] objects; therefore, the client requires a class declaration to produce the stub. The `Server` class that servers as an API to access an object server provides a method `resolve` that resolves the binding between the client stub and the server object.

Iclient/Iserver provides both implicit and explicit mechanisms for resolving references to server objects. In the implicit model, the client code does nothing to resolve references to server objects. The client merely connects to an object server, and starts using well-known objects. To return to the `Counter` example, suppose a client runs the following code:

```
package require Iclient
set serv [iclient::Server #auto \
  sercial.micro.lucent.com 8066]
foo bump
```

The client has connected to an object server but has done nothing to link the server object `foo` to the client. The client proceeds anyway, calling the `bump` method of `foo`. The Tcl parser looks for a command named `foo` but cannot find one defined. Ordinarily, the parser would raise an error saying that the command `foo` does not exist. But Iclient has changed the way Tcl handles unknown commands by introducing its own handler into the built-in `unknown` command.

This handler knows about the object servers that the client is attached to, and it asks each object server if it can `identify` the name `foo`.

```
$server ask identify foo
```

The `ask` command is a primitive to Iclient and Iserver that allows a process to send a Tcl command to another process and wait for the result. In this case, the client sends the script "`identify foo`" to the server, who will evaluate the `identify` command. This command looks to see if `foo` is either the name of a distributed class or a distributed object on the server. The server determines that `foo` is an object, and sends a reply to be evaluated in the client:

```
respond 1 0 "object Counter"
```

The respond command is used to tell the client that the server has a formulated a response to one of its `ask` requests. It reads the return code and the return value from the server, and learns that `foo` is an `object` of class `Counter`. The client knows that `foo` is a `Counter` object, and tries to create it on the client side:

```
Counter foo
```

Unfortunately, the client has also never seen the command `Counter` before. The `unknown` command runs again, this time looking for `Counter`, and asks the server if it knows of a name `Counter`:

```
$serv ask identify Counter
```

This time the server replies that `Counter` is a class. The client asks the server to supply a class stub for the `Counter` class:

```
$serv resolve class Counter
```

```
Counter foo
```

The server must now generate a stub for the class `Counter`, and pass the stub for the client to use. This involves creating a new class declaration that has only the public methods of `Counter`. These stub methods are not implemented as they are in the client class declaration; instead, each is a wrapper around an interface that runs the same method on the server object. The stub class declaration created for the `Counter` class might look like this:

```
class Counter {
  method bump {args} {
    return [eval $server ask object \
      invoke [namespace tail $this] bump \
      $args]
  }
  method value {args} {
    return [eval $server ask object \
    invoke [namespace tail $this] value \
    $args
  }
  ...
}
```

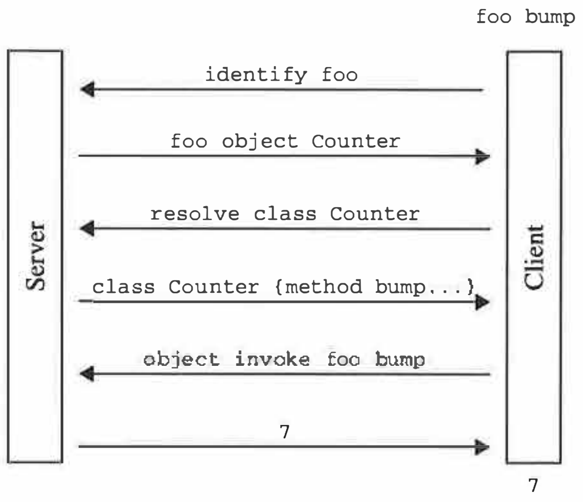
Now the client has a stub for the `Counter` class. It can now try to resolve a reference to `foo`.

```
$serv resolve object foo
```

The client asks the server to help it resolve a reference to foo. This causes the client to create a client stub called `foo` using the class declaration previously generated by the server. With the stub in place, the client can now invoke methods on the stub; when it does so, the stub will ask the server to invoke its own object named `foo` and will use the response from the server as its return value.

This rather detailed chain of events is illustrated in the figure below. What is important to realize is that all this detail was handled by the `Iclient` package; the client script set this into motion by the innocent looking call of `foo bump`. The entire process of resolving is transparent to the programmer, who needed only to create an object on the server, and use that object on the client.

The advantage to this system is that the client code does not have to declare server classes or objects explicitly before using them. Iclient performs late binding to the server objects, doing it only when necessary, and handles all the details of class and object stub generation. But implicit resolving requires Iclient to search through all the object servers connected to a client--usually this



Highlights of Implicit object binding;
Client accesses Server object `foo` for the first time.

represents only a single server, but that does not have to be the case.

To solve these problems, Iclient also offers an explicit interface for generating client stubs. A client can request a stub from a `Server` object, by calling that object's `resolve` method.

```
set serv [Server #auto sercial 8066]
$serv resolve class Counter
$serv resolve object count mycount
```

Above, a client has created a `Server` object to connect to an object server. It then resolves two references, one to the server class `Counter`, and another to the server object `count`. The process of resolving references is the same from this point on; the server helps the client to generate a class declaration and a client stub. The difference is that the client script programmer has explicitly resolved the references before trying to access server objects.

Iclient/Iserver primitives

Iclient/Iserver relies upon a handful of primitives to organize message passing between clients and servers. These primitives include `ask`, `tell`, and `respond`. Whenever a client communicates with a server, or a server communicates with a client, one or several of these primitives are used.

For example, when a client asks a server to create a stub, the client sends a command to the server including the request. The `ask` primitive allows the client to send a Tcl command to the server, evaluate that command on the server side, and learn the result. It works by sending a Tcl command through the client's socket to the server, and supplying the server with a callback. The server uses a fileevent to notice that there is traffic from the client, and reads the command off the socket. It evaluates the command within the limited context of a safe interpreter, and generates a return value. To report the result of the operation, the client sends the client's callback over the socket, along with the server's result. The client reads the callback, evaluates it in its own safe interpreter, and returns the server's result.

```
respond 1 0 "object Counter"
```

The respond command is used to tell the client that the server has a formulated a response to one of its `ask` requests. It reads the return code and the return value from the server, and learns that `foo` is an `object` of class `Counter`.

A third primitive, `tell`, allows a command to be run asynchronously on the other side of the socket. For example, a client can `tell` a server to invoke the `bump` method on object `foo`. The client won't wait for a response; the command will run on the server side whenever the server has a chance to process it. Meanwhile, the client has moved on to other things.

Iclient/Iserver makes heavy use of core Tcl sockets, fileevents, safe interpreters, and the unknown handler to implement distributed objects. Other sources [10] discuss how to connect two processes with Tcl sockets so that each can send and evaluate Tcl commands on the other. Iclient/Iserver follows this approach, and extends it for use with objects.

The server evaluates commands sent from its clients in a safe interpreter, to prevent malicious clients from evaluating destructive commands in the main interpreter. Imagine if a client ran a troublesome command:

```
$server ask exec rm -rf .
```

The server evaluates the command `exec` in its safe interpreter, but `exec` is not defined there. Instead of happily eating up the filesystem, the command raises an error, and the server is safe from harm.

All of the commands that clients call to identify, resolve, create, manipulate, and destroy server objects are defined in the main interpreter, to allow these commands to access the objects there. To allow the clients to call them, the server creates aliases in the safe interpreter, so that when a client asks to run the command `identify`, it does so in the main interpreter of the server.

Application: Hearts

In this section, we will build a game of Hearts that allows four players across a network to meet and play the card game Hearts.

Hearts is a card game usually played by four or more players around a table. Our game will allow players to meet up at a hearts server running in some well-known location, choose a game to join, and then play a game of Hearts. Each player will need to have his own interface with which to play the game.

A client/server approach is natural for this application. There are multiple players, each of whom have their own cards that they want to hide from the others. Each player will be running their own client to play the game, but will need to coordinate their play with the other clients. A Hearts server will manage the deck of cards, synchronize the actions of the clients, and control the flow of the game.

One possible approach to implementing this application is to build a stand-alone version first, that runs on a single machine and lets players take turns playing. There are a lot of details to organize for each game of Hearts--players playing cards, checking their score, organizing their hands--and the engine needs to keep track of what cards have been played, decide whether a player has selected a valid card, etc. On top of this, there are many concurrent games of Hearts being managed by the same server. This problem is neatly solved using an object approach, to help encapsulate each of the Hearts games being played, and to establish a convenient interface between the client and server parts.

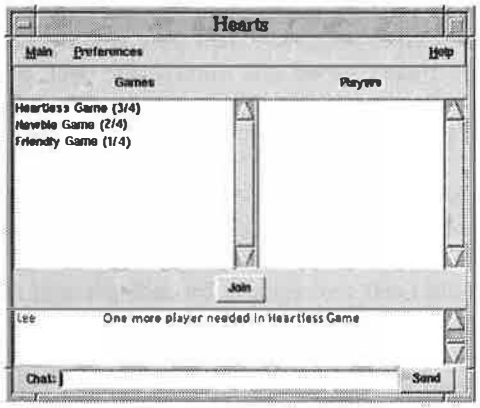
The `Gamemanager` class manages `Game` objects, helping clients to create, join, and quit new games. It also allows clients to register as players, with names and email addresses.

The server initializes itself by defining all the classes it will use during its lifetime, and by creating a well-known `Gamemanager` object, `gm`.

```
package require Iserver
class Gamemanager {
  method games {}
  method players {}
  method newplayer {option args}
  method newgame {option args}
}
class Game {
  method players {}
  method join {}
  method quit {}
  method cards {}
}
...
Gamemanager gm
iserver::listen 8181
```

This is the main script for my Hearts server application; the bulk of the work comes in creating the classes that the server will use to manage games of Hearts.

When a client connects to the Hearts server, it displays a GUI showing descriptions of the various games in session, and the players taking part in the selected game. To learn this information, the client uses the well-known `Gamemanager` object `gm`.



Heartless Game (3/4)
Newbie Game (2/4)
Friendly Game (1/4)

Lee   One more player needed in Heartless Game

Hearts client game selection screen

The following block shows how a client might fill a listbox `.games` with description of the `Game` objects that the `Gamemanager gm` knows about. The binding places the selected `Game`'s players in the listbox `.players`. The button `.join` asks to `join` a `Game`.

```
proc show_games {} {
  global games
  .games delete 0 end
  set games [gm games]
  foreach game $games {
    set desc [$game description]
    .games insert end $desc
  }
}
proc show_players {game} {
  .players delete 0 end
  set players [$game players]
  eval .players insert 0 $players
}
bind .games <ButtonPress-1> {
  set idx [.games curselection]
  set game [lindex $games $idx]
  show_players $game
}
button .join -text "Join" -command {
  $game lock {$game join}
}
```

This block of client code begins by asking the `Gamemanager` object `gm` for a list of `Game` objects. The names of the game objects will be rather dull strings like `game1` or `game7`. The client calls the `description` method of each `Game` object to get a short description of the `Game`, placing this in the listbox `.games`. When the user selects a description in the listbox `.games`, the binding calls the `players` method of the corresponding `Game` object, and uses those names to fill the `.players` listbox.

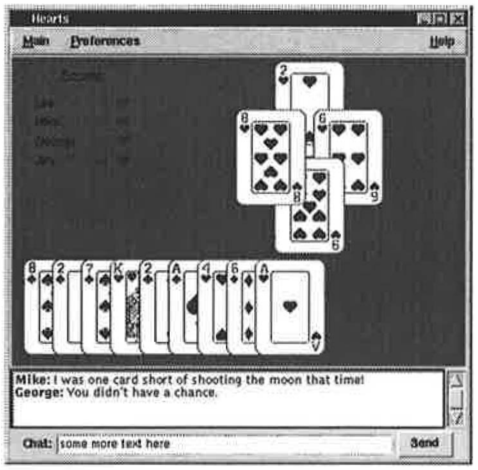
The `.join` button allows a player to join a `Game` that is forming. Since this alters the state of that `Game`, the button command requests a lock before joining. All the players in that game will want to know that a new player has joined. They automatically learn this by `watching` the `Game` object; when the lock is requested, the clients ask the `Game` object for the new list of players, and update their listboxes.

```
$game watch lock {
  show_players $game
  show_games
}
```

Now the GUI changes as players enter and leave games. The client also monitors the Gamemanager to `watch` for `Game` objects being added:

```
gm watch lock {
  .games delete 0 end
  set games [gm games]
  foreach game $games {
    .games insert end [$game description]
  }
}
```

Suppose our player has selected a `Game`, and it has filled with players. It is now time to begin a game of Hearts. The view of the available games is replaced with a canvas showing the game table, with the player's cards at the bottom, and a view showing the cards that others have played in the center. The player can see the current score of the game at the upper-left comer. A chat box at the bottom allows the player to send messages to the other players in the game.



Mike: I was one card short of shooting the moon that time!
George: You didn't have a chance.

some more text here

To generate the hand, the client calls the `cards` method of a `Game` object. Assume for our purposes that the name of the `Game` object for this game is stored in the variable `game`. The following client code requests a list of cards in the player's hand, and draws them on the canvas by calling a command on the client.

```
card_draw [$game cards]
```

When the player is ready to play a card, they drag the card to the center of the table. The game of Hearts is played by following an order of play. One player leads by playing a card on the table, then play moves around the table in a circle. If our client tries to play a card, the server must first decide if it is that player's turn.

The `Turn` object keeps track of the order of play for a given trick, and prevents players from playing out of turn. Players invoke the `play` method of the `Turn` object in order to play a card.

To do this, the client requests a lock on the `Turn` object. The `Turn` class exists primarily to regulate whose turn it is, and to allow only the appropriate player to play a card. To accomplish this, the Turn object establishes a set of locks on itself for each of the players playing the game. These lock requests enter a queue, and are entered in turn order. If a client tries to access the turn object prematurely, then that client must wait for the players ahead in the play order to play their cards; as soon as they are finished, then our client acquires the lock and plays a card.

Client Code

```
$turn lock {
  $turn play DK
}
```

Server Code

```
class Turn {
  inherit ::iserver::Distributed
  private variable order;
  method newturn {}
  method discard {}
  private method accept {card}
  method play {card} {
    set player [::iserver::current]
    set turn [lindex $order]
    if {$player == $turn} {
      accept $card
      report $discard
      return
    } else {
      errpr "Not your turn"
    }
  }
}
```

The `Turn` object controls the order of play. With every trick, the `Turn` object calculates the order of play based on the rules of Hearts. When a player attempts to `play` a `card`, the `Turn` object asks Iserver who this client is, and checks to see if it is the client's turn. If it is, the `Turn` object `accepts` the `card`, and reports a `discard` event to let the other clients know to redraw their canvases. If the player should not be able to play yet, the `Turn` object does not let the player `play`.

All of the clients are going to be interested in the cards that the other players have played, in order to draw them in the center of the canvas. To watch for players playing cards, each client watches the `Turn` object. When another client accesses the `Turn` object to play a card, the server notifies each of the clients, who then learn which cards were played and draw them:

```
turn watch discard {
  set cardsPlayed [$turn discard]
  discard_draw $cardsPlayed
}
```

Notice that the script a client runs in response to watch event can do anything; it does not need to limit itself to accessing just the object being monitored. The client is watching the turn object to see when a card is played; once it is notified, it calls the `discard` method of the `Turn` object to learn which cards were played, and then draws them using the `discard_draw` command.

There are situations in which it is more convenient for a server to broadcast commands to its clients than for the clients to register interest in an event using `watch`. Players communicate with one another by using an object of class `Chat` that lives on the server. When players wish to talk to the others in the game, they call the mesg method of the Chat object, and pass their message along. The easiest way for the Chat object to communicate the message along to the clients is to keep a list of listening clients, and then remotely invoke a procedure on the client side to

Server Code

```
class Chat {
  method listen {} {
    set client [::iserver::client]
    ::iserver::list_add listeners \
      $client
  }
  method talk {mesg} {
    set speaker [::iserver::client]
    foreach client $listeners {
      iserver::tell $client rpc \
        chat_mesg $speaker $mesg"
    }
  }
  variable listeners
}
```

Client Code

```
$serv rpc chat_mesg {speaker mesg} {
  .chat.text insert end $speaker $speaker\
    "\t$mesg" $speaker
  .chat.text see end
}
```

With distributed server objects, locks, and watch callbacks, the task of organizing a multi-player card game is made easier.

Conclusions

Using class methods as a natural interface between clients and server, distributed server objects allow efficient encapsulation of data and operations.

It allows encapsulation of data and operations and uses class methods as a natural interface between clients and server.

Iclient/Iserver fills a niche between simple Tcl-based remote procedure call systems like Tcl-DP and comprehensive object brokers that follow the CORBA standard. It provides an easy way to construct client/server applications while eliminating the need to write a separate set of remote procedure calls. Distributed server objects, visible and accessible by all connected clients, provide a natural interface for sharing information. The locking and object watching services provide synchronization and event notification to avoid concurrency headaches, including the need to poll continuously to see if the server has changed.

Using Iclient/Iserver protocols, starting to create client/server applications is easy, requiring trivial changes to the server objects that will be distributed and a few lines of code on the server and client ends to establish a connection. After that, the client resolves the objects it wishes to use, and invokes server object methods through its own client stubs.

While the approach is convenient, it can generate a lot of server traffic, since every remote method call requires a network transaction. Future work will examine how best to allow clients to cache additional information in their client to avoid talking to the server to perform read-only operations. This will obviously involve increased complexity in designing classes, and may raise a new set of concurrency issues, but it stands to improve performance and scalability where many clients access a single server.

Future work will also improve security in the system, to allow servers to authenticate connecting clients. Present security is achieved by evaluating socket traffic through a safe interpreter, which restricts the entry-points to the server to the methods of its distributed objects. If the server could authenticate its clients, then it could provide distributed objects that performed more trusted operations with greater security. Finer control over the methods that a given server object publishes to a specific client could also help to create more flexible server classes.

With version 3.0, [incr Tcl] has become a pure extension to Tcl/Tk, allowing vanilla interpreters to use classes and objects by loading [incr Tcl] as a package. With this support, these developers who enjoy the flexibility of scripting and the structure of an object system can use [incr Tcl] without having to build their own custom interpreters. By loading Iclient/Iserver, these same developers can use their existing classes as interfaces and construct client/server applications faster and more easily than ever before.

Acknowledgments

Thanks to Michael McLennan and George Howlett for many useful discussions about client/server architectures and distributed objects, and for their help with this paper. Thanks also to Michael McLennan for his help in designing the interface for Iclient/Iserver software. Finally, thanks to Robin Valenza for losing sleep to edit this paper.

References

[1] Jacob Levy, "A Tk Netscape Plugin," *Proceedings of the Fourth Annual Tcl/Tk Workshop '96*, Monterey, California, July 10-13, 1996.

[2] Michael J. McLennan, "[incr Tcl]: Object-Oriented Programming in Tcl," Proceedings of the Tcl/Tk Workshop, University of California at Berkeley, June 10-11, 1993.

[3] Michael J. McLennan, "The New [incr Tcl]: Objects, Mega-Widgets, Namespaces and More," *Proceedings of the Third Annual Tcl/Tk Workshop '95*, Toronto, Ontario, Canada, July 6-8, 1995.

[4] Robert Orfali, Dan Harkey, and Jeri Edwards. The Essential Distributed Objects Survival Guide. John Wiley and Sons, 1986. ISBN 0-471-12993-3.

[5] http://www.omg.org

[6] B. C. Smith, L. A. Rowe, and S. Yen, "Tcl Distributed Programming," *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, Jule 10-11, 1993.

[7] Peter T. Liu, Brian Smith and Lawrence Rowe, "Tcl-DP Name Server," *Proceedings of the Third Annual Tcl/Tk Workshop '95*, Toronto, Ontario, Canada, July 6-8, 1995.

[8] M. Roseman and S. Greenberg, "Building Real Time Groupware with GroupKit, a Groupware Toolkit," *ACM TOCHI*, March 1996.

[9] Mark Roseman, "Managing Complexity in TeamRooms, a Tcl-Based Internet Groupware Application," *Proceedings of the Fourth Annual Tcl/Tk Workshop '96*, Monterey, California, July 10-13, 1996.

[10] Mark Harrison, Michael McLennan. *Effective Tcl/Tk Programming.* Addison-Wesley, 1998. ISBN 0-201-63474-0.

Data Objects

George A. Howlett
Bell Labs Innovations for Lucent Technologies
gah@bell-labs.com

Abstract

Scripting languages are great for gluing together components, but they suffer as the complexity or size of data scales upward. Data objects solve this problem by marrying both high-level and low-level programming styles. Data objects are self-contained representations of data. They define both the structure of the data and the methods to access it. Data may be accessed through both Tcl and a C interface. This paper will describe two such data objects, a vector and table object.

Introduction

Scripting languages such as Tcl[1], have been described as fundamentally changing the way people write programs, representing a very different style of programming[2]. They let developers rapidly form new applications from components and pieces of existing applications. What distinguishes scripting languages from conventional structured programming languages (e.g. C, C++, FORTRAN) is that they are high-level, interpreted, and weakly typed.

Despite their advantages, scripting languages do not replace structured programming languages. They, in fact, offer a very different set of trade-offs.

Where Scripting Languages Succeed

Scripting languages make it easy for programmers to control how components are used. Because programming is done at a high-level, components have simple interfaces.

Components are identified by strings, not pointer addresses. Even heterogeneous components can be referenced and grouped by a simple list of names. Components also have well-defined operations that can be used to access or modify their internal data without requiring an understanding of how the data is structured.

Scripting languages are usually interpreted, so the flow of program execution is simple to change and test. It's easy to wire applications together quickly or try out new algorithms.

A good example of high-level components is the Tk widgets. Widgets are referenced by string identifiers. They have well-defined **`configure`** and **`cget`** operations that change widget attributes. (You can change a widget's font without knowing anything about the structure of a Tk font). It's easy to rearrange widgets or add and test new behaviors using the **`bind`** command.

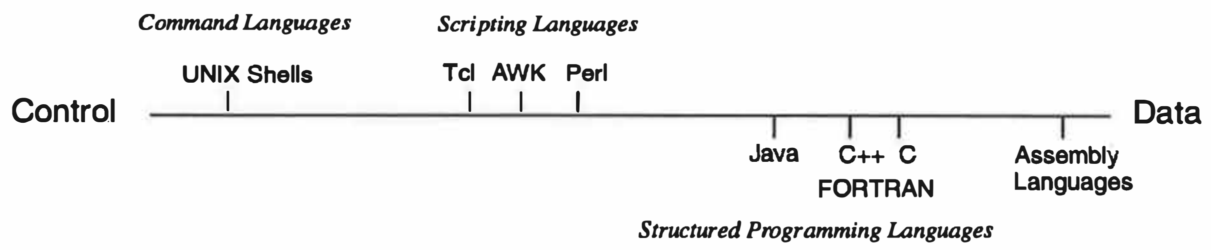
Where Scripting Languages Fail

Scripting languages are inversely weak where structured programming languages are strong.

As an application's data scales from tens to thousands of elements, scripting languages perform worse than compiled low-level languages. Both in terms of performance and memory utilization, scripting languages carry greater overhead. For example, a simple array of 10,000 elements in Tcl will use as much memory to store the indices as the values. Arrays in C have no such overhead.

Flexibility has a cost. Weak typing requires that data many times be converted back-and-forth from strings to native machine types (e.g. ints, doubles). For example, the time to plot a graph of 10,000 data points is dominated by the string-to-decimal conversions, not the performance of the X server.

Figure 1. Programming Languages: Control Versus Data



Scripting languages lack facilities for structuring and manipulating complex data. They are, by definition high-level. You can not easily define new data types, access pointers, or manage native types.

Side-effects of weak data structuring can be seen in Tk. Widgets often get used as data containers. For example, string data is sometimes put into a text widget just to use its more powerful search and replace operations.

Data also becomes tightly coupled to widgets. Let's say you are displaying file directory information in a listbox widget. You may also maintain several Tcl arrays for various information fields (owner, date, size, etc.) other than the file name. Each of these data containers must be synchronized. If entries are inserted or deleted or the listbox is itself destroyed, the other data containers must be likewise updated. Program control becomes increasingly complex and error prone as more parallel data structures are required.

What Makes a Language High Level?

It is usually in terms of data where scripting languages fail. In applications data can scale much more quickly than code size, sometimes by several magnitudes. High level data abstractions preclude fine-grain access to data. Furthermore, the overhead of weak typing both increases memory consumption and slows data processing.

Consider programming languages in terms of how finely or coarsely they handle data. Figure 1. shows a continuum from high-level command languages to low-level structured programming languages. High level languages abstract data to remove details, making control very simple. Structured programming languages provide a very fine-grain control of data. Conversely, control is complicated by the details and complexity of the data representations.

Data Objects

Tcl was originally designed to be extensible, allowing you to link your own C code to extend the language by adding new commands and variables. We can use this feature to represent data at both programming levels, as *data objects.*

Data objects are simply containers for data. They are objects because each instance represents both a set of data and high-level operations for querying and modifying that data[†].

Programming at Both Levels

At the Tcl programming level, a data object is referenced by its string identifier. Objects can be shared simply by passing their names. Object data is accessed or changed through a collection of well-known operations. Because data operations are written in C code, operations for data objects run as fast as compiled code.

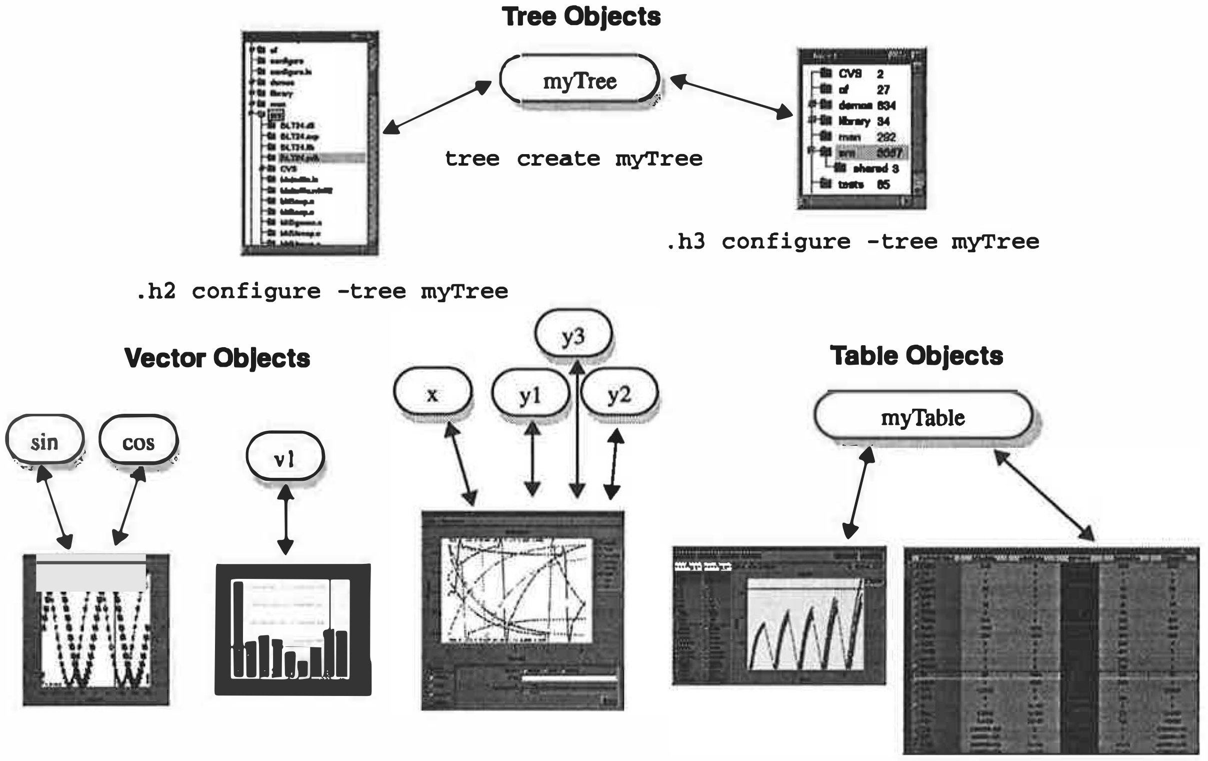
Data objects also provide a low-level programming interface. An object's data can be accessed directly from its C API, without going through Tcl or converting the data from a string. For example, widgets may use this interface to access the data in its native format.

New Direction: Data Object Based System

Imagine a system where data objects are ubiquitous. In this hypothetical system, data objects are the new currency for building applications. Standard data objects automatically work seamlessly with widgets. Extensions plug together. A database can automatically display its data in a spreadsheet widget because they both use the same table data object.

†. Data objects are object-based. This differs from object-oriented in that data objects have no inheritance or ability to define new operations.

Figure 2. Data Objects in BLT



The point is not that people want to write low-level code. On the contrary, if standard data objects became widely used, many applications could be written by plugging several large components together.

The BLT toolkit has slowly moved in this direction, representing complex data not as strings but by data objects. BLT currently has two data objects: a vector object representing an array of numbers, and a tree object representing a hierarchy. A table object will also soon be available. Several widgets in BLT work with data objects as shown in Figure 2. For example, the **hierbox** widget can use the tree object for its data. Note that more than one widget can use the same data object. Each widget can in turn offer a different view of the same data.

The following sections will describe both a concrete example of a data object (the vector object) and a real-life application built using data objects.

Example: Vector Objects

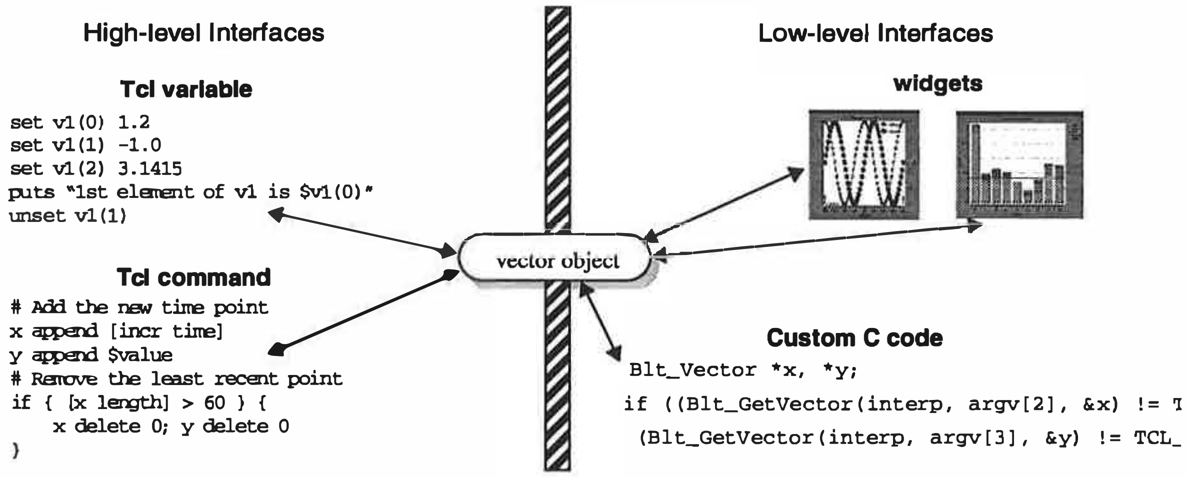
Originally the only way to pass X-Y coordinates to the BLT plotting widgets was as Tcl lists (`-xdata` and `-ydata` options). Internally, the graph processes the string data, converting it into an array of doubles.

```
# Data values
set x { 0.2 0.4 0.6 0.8 1 1.2 }
set y1 { 11 21 28 34 38 39 }
set y2 { 26.2 50.5 72.9 93.3 112 128 }
barchart .b

# Add two new elements to the graph
.b element create e1 -xdata $x -ydata $y1
.b element create e2 -xdata $x -ydata $y2
```

This is inefficient in terms of memory. The same data points are stored in two different formats: Tcl lists and binary data. It's also inefficient in terms of performance. It's not usual to plot thousand of data points. New coordinates may also be added over time. Each time new data points are added to the list, the entire list must be converted from decimal strings to double precision values.

Tcl arrays can't be used because there is no implied ordering to associative arrays. Especially for plotting, you need to insure the second data point comes after the first, and so on. This isn't possible since arrays are really hash tables. Also, associative arrays consume memory because both the index and value are stored as strings for each data point.

Figure 3. Vector Object Interfaces



```
set v1(0) 1.2
set v1(1) -1.0
set v1(2) 3.1415
puts "1st element of v1 is $v1(0)"
unset v1(1)
```

```
# Add the new time point
x append [incr time]
y append $value
# Remove the least recent point
if { [x length] > 60 } {
    x delete 0; y delete 0
}
```

```
Blt_Vector *x, *y;
if ((Blt_GetVector(interp, argv[2], &x) != T
 (Blt_GetVector(interp, argv[3], &y) != TCL_
```

A worse problem is that data is tightly coupled to the widget. Data selection and analysis are important parts of plotting. Every data operation requires coordinate data to be translated from Tcl strings.

One alternative to is add selection and analysis functions to the widget itself (e.g. text widget). It seems the "essential" set of functions is always increasing. The danger is that as the widget grows in complexity and code size, it becomes monolithic. The focus of the widget shifts from displaying data to managing it.

Vector Data Object

Instead the **graph** and **barchart** widgets use vector data objects. A vector object simply represents an array of doubles, indexed by integers. The `-xdata` and `-ydata` options of the graph recognize vector names. The graph directly accesses the vector object's data in its native format.

```
vector create x y1 y2
x set { 0.2 0.4 0.6 0.8 1 1.2 }
y1 set { 11 21 28 34 38 39 }
y2 set { 26.2 50.5 72.9 93.3 112 128 }
barchart .b
.b element create "e1" -xdata x -ydata y1
.b element create "e2" -xdata x -ydata y2
```

Vector objects have a variety of interfaces. They are shown in Figure 3. From Tcl, vector object data is accessed through a variable or command. Vector objects also can be extended by user-defined C code.

Vector objects are created with the **vector** command.

```
vector create v1
```

A new vector `v1` is created. At the same time, both a Tcl command and Tcl array variable `v1` are also created.

Vectors can be used like Tcl arrays. Vectors are indexed by integers, starting from zero. When elements of the array are read, set, or unset, the corresponding elements in the vector are accessed.

```
set v1(0) 1.2
set v1(1) -1.0
set v1(2) 3.1415
puts "1st element of v1 is $v1(0)"
unset v1(1)
```

The advantage of mapping an array to the data object is that it makes vector data appear and act like ordinary data in Tcl.

The vector's Tcl command can be used to access or modify elements, invoking one of several operations. For example, the **delete** operation removes elements by their index.

```
# Delete the first element
v1 delete 0
```

The **expr** operation performs arithmetic on vector.

```
v1 expr { v1 * (v2 + 10) }
v3 expr { sin(v2)^2 + cos(v1)^2 }
```

It's important to note that data objects provide *high-level* operations. This is different than translating low-level C or C++ code into Tcl. For example, to find numbers that lie in a certain range, I could write a loop that builds a new list. In fact, the code isn't very different than if this was programmed in C.

Listing 1. Example of Vector Object C API

```
Blt_Vector *xVec, *yVec;
int i, length;
double *x, *y;

/* Get the two vectors to multiple.*/
if ((Blt_GetVector(interp, argv[2], &xVec) != TCL_OK) ||
   (Blt_GetVector(interp, argv[3], &yVec) != TCL_OK)) {
    return TCL_ERROR;
}
length = Blt_VecLength(xVec);
/* Check that the vectors are the same length */
if (length != Blt_VecLength(yVec)) {
    Tcl_AppendResult(interp, "vectors ", argv[2], " and ", argv[3],
   " have different lengths",(char *)NULL);
    return TCL_ERROR;
}
/* Allocate result array and compute the product */
array = (double *)malloc(legnth * sizeof(double));
x = Blt_VecData(xVec), y = Blt_VecData(yVec)
for (i = 0; i < length; i++) {
    array[i] = x[i] * y[i];
}

/* Update the vector so it knows that its data has changed.
Old data will* automatically be freed. */
if (Blt_ResetVector(yVec, array, length, length, TCL_DYNAMIC) != TCL_OK) {
    return TCL_ERROR;
}
return TCL_OK;
```

```
set len [v1 length]
for { set i 0 } { $i < $len } { incr i } {
    if {($v1($i) < $lo) && ($v1($i) > $hi)} {
        lappend values $v1($i)
    }
}
```

The vector object instead provides a simple, high-level **search** operation.

```
set values [v1 search -value $low $high]
```

Vectors can be converted to and from strings (lists). The **set** operation sets the values of the vector from a list. The **range** operation returns a list of vector elements between two indices.

```
# Set the vector from a string.
set list { 1.1 0.0 2.3 4.4 -1.0 }
v1 set $list
# Set the string from the vector.
set list [v1 range 0 end]
```

C API

No matter how many built-in operations a data object may have, it's likely that some functionality will always be missing. Data may need to be read in from a specific file format. Special calculations may be required. Therefore objects themselves need to be extensible.

A vector object's data can also be accessed from C using its library API. In the same way that Tcl is extensible, new code can be written to manipulate vectors in ways not available from its Tcl interface For example, the **spline** command in BLT uses vectors to create interpolating splines. Listing 1. shows an example of the C API to multiple to vectors together.

Vector Notifications

Vectors provide a hook or callback to notify clients (such as the graph or barchart) when the vector data changes. Notifications usually as occur as idle tasks, but this can be user-controlled.

Notification occurs automatically, no matter how the vector was changed: via the vector's Tcl command, array variable, or C API. You can therefore separate the data processing portion of your application from the GUI.

For example, a graph that displays only the last 60 time points can be built using a barchart and a pair of vector data objects to hold the X-Y coordinates. As new data arrives, the new time point is appended to the `x` and `y` vectors. If there are more than sixty time points, the oldest is removed.

```
# Add the new time point
x append [incr time]
y append $valuea
# Remove the least recent point
if { [x length] > 60 } {
    x delete 0; y delete 0
}
```

There is no code to synchronize the barchart. It's not needed. The chart is redrawn automatically. The barchart sees the new values because it shares the vector's data instead of making its own internal copy. While several graphs can use the same vector, there will be only one copy of the data.

Vector Performance

Since vectors are very simple data types, they can be compared with Tcl lists.

Numeric values were read into a variable **data** and arithmetic was performed on those values. The vector-based example *vector.tcl* is listed below.

```
vector create a b c d x
a set $data
b set $data
c set $data
d set $data
x expr { a * b + c * d }
x expr { a + b * c + d }
```

For lists, both byte-compiled (8.0) and pure-interpreted (7.6) versions of Tcl were tested. Vector operations are written in C code, so they are unaffected by the byte-compiler. The following example *list1.tcl* replicates the vector example, this time using lists. The procedures **add** and **mult** perform arithmetic on lists.

```
proc mult { list1 list2 } {
    set len1 [llength $list1]
    set len2 [llength $list2]
    if { $len1 != $len2 } {
        error "lists are different lengths"
    }
    foreach e1 $list1 e2 $list2 {
        lappend result [expr $e1 * $e2]
    }
    return $result
}
```

```
proc add { list1 list2 } {
    set len1 [llength $list1]
    set len2 [llength $list2]
    if { $len1 != $len2 } {
        error "lists are different lengths"
    }
    foreach e1 $list1 e2 $list2 {
        lappend result [expr $e1 + $e2]
    }
    return $result
}
set a $data
set b $data
set c $data
set d $data
set x [add [mult $a $b] [mult $c $d]]
set x [add [add $a [mult $b $c]] $d]
```

Another version, *list2.tcl*, is included below. The only difference is that the **add** and **mult** routines are replaced by a generic procedure **calc**. The operator is passed as an argument.

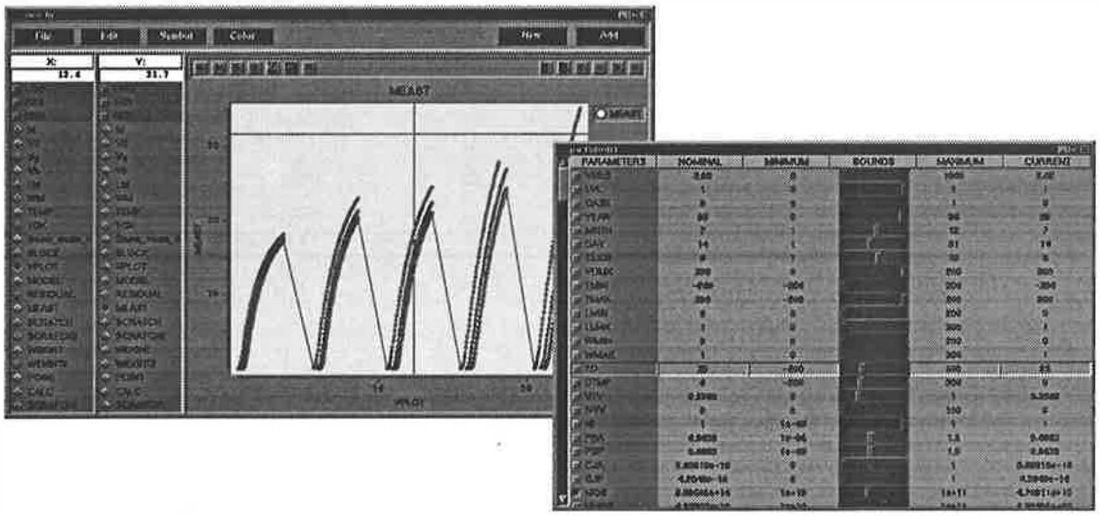
```
proc calc { list1 op list2 } {
    set len1 [llength $list1]
    set len2 [llength $list2]
    if { $len1 != $len2 } {
        error "lists are different lengths"
    }
    foreach e1 $list1 e2 $list2 {
        lappend result [expr $e1 $op $e2]
    }
    return $result
}
set a $data
set b $data
set c $data
set d $data
set x [calc [calc $a * $b] + [calc $c * $d]]
set x [calc [calc $a + [calc $b * $c]] + $d]
```

Each test was run using an increasing number of values. The time (in seconds) for each test is listed below.

	Number of Values		
	1,000	**10,000**	**100,000**
7.6 *list1.tcl*	0.60 secs	5.3 secs	59.5 secs
7.6 *list2.tcl*	0.63	5.6	62.3
8.0 *list1.tcl*	0.27	1.7	21.4
8.0 *list2.tcl*	0.90	8.0	85.3
vector.tcl	0.16	0.6	10.5

The improvement of (byte-compiled) Tcl 8.0 over Tcl 7.6 is striking. Not surprisingly though, vectors out perform both the byte-compiled and non-compiled versions

Figure 4. Snapshot of Camelot Parameter Extractor



of Tcl using lists‡. Vector operations written in C code will generally have less overhead than byte-compiled code. For *vector.tcl*, the read in and string-to-decimal conversions were the greatest portion of the time spent, not the arithmetic.

Application: Parameter Extractor

A parameter extractor compares the behavior of a model of an IC circuit to measured results from real devices. A model is a set of equations (written in a structured programming language like C or FORTRAN) that given input voltages, returns an output current. Models also have parameters, such as temperature and device size, that affect its calculations.

The job of a parameter extractor is to evaluate the model of a circuit repetitively, bumping parameter values up and down in a way that minimizes the error between the model's outputs and measured data. The end product is a set of model parameters that best reflect the real devices.

This a computationally hard task. There can be thousands of measured data points. Models can have scores of parameters, allowing many degrees of freedom. Complex or over-parameterized models may exhibit multiple local minima. One method to stabilize the extraction process is to limit the number of the parameters that can be altered. It may also be desirable to constrain the parameters to lie in certain ranges (e.g. a resistance must be positive). How one selects initial parameter values, their constraints, and representative data regions matter greatly.

‡. The results of *list2.tcl* demonstrate the penalties of dynamic code. Because the operator (* or +) was passed as an argument to the `calc` procedure, the Tcl 8.0 interpreter was unable to fully compile the procedure. As a result, it is slower than the Tcl 7.6 interpreter.

High-level Programming?

Such compute-intensive tasks are usually the domain of low-level structured programming languages. The extraction step requires speed and numeric precision. But data selection and filtering is an equally important aspect of the application. The extractor must manage the various inputs (voltage, temperature, etc.) and the outputs (currents returned from a particular circuit). Additionally the model parameters must also be coordinated. Scripting languages are better at controlling how and where data is used. For parameter extraction, this is especially useful to experiment with different heuristics or wire in new models.

Camelot Parameter Extractor

The Camelot parameter extractor developed at Bell Laboratories, lives in both programming worlds. Figure 4. shows a snapshot of the application. The currency of the application is a table data object.

The table data object represents a dynamically resizeable table of values. Rows and columns of the table can be selected, sorted, duplicated, etc. Each row and column has its own label and can be used just as a vector. For example, you can plot different columns of values.

The code below creates a new table object d0. A new Tcl command d0 is also created that can be used to access the table of data from Tcl. Table objects have several generic operations. The **read** operation reads tabular ASCII data from a file into the object. The **column extend** operation adds a new column to the table. The **column label** operation sets the label for columns.

```
table d0
d0 read sh.data
d0 column extend 1
d0 column label 2 VGS
d0 column label end IDSHAT
```

During the extraction process the model will write its outputs into the table. Models can be evaluated thousands of times during an extraction step. The outputs are later compared against the measured outputs to determine the fit. For example, a MOS model may store the modeled currents (I_{DS}) in the last column of the table. Models are quickly evaluated because the table object's data structure is immediately available through a C API.

Filtering and selecting of data are done from the table's high-level programming interface††. The **select** operation, chooses all rows that match a vector expression. In this case, we are looking for rows where column VGS is equal to $value.

```
d0 row select { VGS == $value }
d0 dup d1
```

The **dup** operation creates a new table object d1 that contains only the selected rows of d0.

Parameters are stored in a second table object. It contains various parameter information; the initial and current parameter values, the minimum and maximum bounds, etc.

A new table object p0 is created.

```
table p0
p0 read sh.pars
```

The parameters file sh.pars is then read into the table. The file contains the following information.

†† It is interesting to note that while custom C code tends to be application specific, the high level Tcl filtering and selection operations are generic.

```
NAME      INCLUDE MIN     MAX      NOMINAL
BETA      0.0     25e-4    1e-4    50e-4
LAMBDA    0.0     0.0     0.0      0.1
VTH       0.0     0.8     0.4      1.2
DELTA     0.0     0.0     0.0      1.0
ATS       0.0     7.0     2.0      40.0
AST       0.0     15.0    2.0      50.0
RST       0.0     0.05    0.000001 1.0
NST       0.0     1.1     1.01     4.0
THS       0.0     0.0     0.0      0.5
THC       0.0     0.0     0.0      1.0
```

The first column is the parameter name. The second column is the inclusion status of the parameter. If the value is 0.0, the extractor will not adjust its value. The third and forth columns are the bounds of the parameter. The last column is the initial parameter value.

To run the extractor, you select data and the parameters that you wish to optimize. The **extract** command runs the extractor using the two table objects. The resulting new set of parameters will be appended as new column in p0.

```
d0 row select { IBL == 3.0 }
d0 dup d1
extract d1 p0
```

Since data objects are easily connected to widgets, the new parameters are automatically displayed in a table widget. The parameter fits are plotted in the graph widget.

The table objects act as a go-between to the high and low-level programming worlds. Like many applications, Camelot benefits from working at both levels.

Conclusion

Data objects act as basic application building blocks. The high-level Tcl interface allows large numbers of objects to managed easily. Built-in operations let the user accomplish most data transformations from the Tcl level. The C level interface allows specialized code to be performed by revealing the object's internal data.

Two examples of data objects are vectors and tables. They both represent common data structures for many applications. The idea or utility of these objects is nothing new. Interactive statistical programming languages such as S[3] or MATLAB have demonstrated the power of vector and matrix programming for many years. What is new is how these objects can be applied to very high-level languages such as Tcl to solve a wide class of performance problems and to simplify interactions with data.

Several questions remain open. What are the standard data objects? What like of low-level API is required? Is a better object system, such as [incr Tcl] required?

The hope is that if a set of objects becomes ubiquitous, it will form the basis for large plug-able components. These components would in turn become the building blocks for greater applications.

References

1. J. K. Ousterhout, ***Tcl: An Embeddable Command Language***, Proceedings of the 1990 Winter USENIX Conference, 1990, pp. 133-146.
2. J. K. Ousterhout, "Scripting: Higher-Level Programming for the 21st Century", *Computer*, March 1998, pp. 23-30.
3. R. A. Becker, J. M. Chambers, A. R. Wilks, *The New S Language*, Wadsworth & Brooks/Cole, Pacific, CA. 1988.

A Tcl-based Multithreaded Test Harness

Paul Amaranth
Aurora Group, Inc.
email: paul@auroragrp.com

Abstract

This paper describes an implementation of a test harness written in Tcl and C for Merit Network, Inc. capable of running multiple structured tests simultaneously. Many available test tools are based on a single threaded stimulus/response approach. This is not always sufficient to adequately test certain classes of applications that manage multiple simultaneous requests. The design presented in this paper is for a Tcl-based system capable of running multiple simultaneous tests in the context of testing a RADIUS authentication server. The design and implementation illustrate novel applications of Tcl as well as the rapid development and code reusability inherent in using Tcl as an application glue language.

1 Introduction

Merit Network, Inc is a nonprofit entity affiliated with the state universities of Michigan that is responsible for supplying Internet services to its member universities and affiliates. As part of fulfilling that mission, Merit Network, Inc. has developed an implementation of an Authentication, Authorization and Accounting (AAA) server based on the RADIUS protocol. Through the free and licensed distributions of their server, as well as participation in the IETF and RADIUS working groups, Merit has been a driving force in the development of the RADIUS protocol. Recently, Merit has been the key player behind the formation of a RADIUS industry consortium formed to foster industry cooperation and shared development using the Merit AAA Server as a base.

The formation of the consortium and the transition from a university based software package to a commercially licensed package brought the realization that current testing procedures were inadequate. In particular, tools were needed to

- Perform regression testing
- Simulate typical load situations
- Provide performance benchmarks

As far as practical, the tools developed would need to meet the following goals:

- satisfy the requirements above
- be simple and convenient to use
- allow for canned and automated scripts

The RADIUS protocol is a simple UDP transaction based client/server protocol. In a typical interaction, an authentication request message is sent from a Network Access Server (NAS) to the RADIUS daemon. The message contains a prospective user's ID and encrypted password. The daemon either performs the authentication function locally, looking the user up in a local file, database or password file, or hands off the request to a remote RADIUS server to be authenticated remotely. In the latter case, while waiting for a reply from the remote server, the initial server may handle other incoming requests. Once authenticated (or rejected), the server returns a reply to the requesting NAS. Subsequently, the NAS may send accounting messages containing billing information on the user's session. These messages must be acknowledged and processed.

The structure of the Merit AAA Server daemon is arranged so that tasks that may take significant time intervals (consulting a database or password file, for example) are forked off to a child process. When the child process completes, it informs the parent of the result and a response is ultimately formulated as a reply. In the meantime, the parent process may accept additional incoming RADIUS messages, which may result in more child processes and so on.

The problems involved in adequately testing software of this nature are difficult and complex. Interactions between multiple processes may result in subtle errors. In many cases, however, it may be adequate to formulate an authentication request or accounting message, send it to the daemon and observe the reply. This at least serves to verify the basic functionality. More thorough testing requires loading the server with multiple requests until it shows signs of stress [Myers, Deutch].

Merit has a simple tool used for general testing purposes called *radpwtst*. This is a single threaded tool that sends a message and waits for a reply. As such, it was not sufficient for use in load or stress testing, although it did serve to verify functionality.

Expanding the use of radpwtst in conjunction with *expect* [Libes], was one approach considered. This did not meet the goals of simplicity, ease of use and multiple ongoing tests. The DejaGnu system [Savoye] solves some of these problems and offers the added advantage of a standard testing framework, but also suffers from the single threaded test limitation. Consequently, the decision was made to build a test harness facility. Design goals included:

- Ability to handle multiple simultaneous tests

This was a key requirement. The ability to launch multiple simultaneous tests and match replies to their appropriate origin allows for load testing and benchmarking. Since any given test instance may involve multiple interactions with the remote server, each instance must maintain its own context and exist in a separate thread.

- Extensibility

It was considered important that new tests should be added without having to modify the test harness itself. Since it is not possible to imagine all possible tests that may be employed, a mechanism was required that would provide a great deal of flexibility in the test structure.

- Simple and convenient to use

It was envisioned that programmers would write test code to exercise their RADIUS daemon code and to verify that bugs were fixed and stayed fixed across releases. For this to be reasonable, it had to be relatively easy to use and require a minimum understanding of the test harness internals. That is not to say writing a test is easy; it still requires knowledge of the code being tested and the details of the RADIUS protocol.

In addition, a secondary factor involved portability. The initial development environment was a Sun platform, but there there was a concern that the test software could be ported with minimal effort should the need arise.

Tcl was chosen as an implementation vehicle primarily for its extensibility, portability and ability to serve as a glue language to bring together pieces of existing code under a new framework. An initial prototype was constructed to provide functionality similar to the existing *radpwtst* tool. A loadable set of Tcl extensions was written that provided interfaces for initializing the data structure utilized by the underlying *radpwtst* routines as well as sending and receiving RADIUS messages.

The prototype proved surprisingly useful and a number of different tests were written using the facility. Embedding in Tcl resulted in a powerful programmable test facility. As an example, passwords could be corrupted on a pseudorandom basis and the expected reject response could be looked for. It was also a relatively simple matter to check for one of the key features of the Merit AAA Server: the management of simultaneous user sessions. In this case, an authentication request for a user should be rejected if they had reached a configurable session limit, otherwise it should pass. Failure to follow this behavior is a serious error. It was a relatively simple task to write a Tcl test procedure that tracked open sessions and reported when either authentication failed when it should have succeeded or vice versa.

The initial success of the prototype was encouraging, but key features were lacking, particularly the management of simultaneous tests, i.e. multithreading. Although more flexible than the radpwtst tool, it was still single threaded. In addition, writing tests involved, in effect, writing an entire Tcl program. Consideration of these issues led to the current design.

2 The Test Harness Design

The design presented is based on a round-robin test scheduler [Tanenbaum, Say] or executive, interpreting test *templates* [Stocks]. Templates specify test actions which are Tcl procedures. Templates are compiled into an internal format and executed by the test executive.

The test harness design is centered around the test specification or template. The syntax for the test template is shown below in extended BNF:

```
<name>: <test>

<test>::=  <setupProc>
           [<delay>] [<postProc>]
  {
  [ F: <test>]
  [ S: <test>]
  }
<setupProc> ::=  <Tcl procedure name>
<postProc>  ::=  <Tcl procedure name>
<delay> ::= <numeric value>
        | $<Tcl global variable name>
        | <open bracket> <Tcl procedure>
          <close bracket>
<open bracket>  ::= [
<close bracket> ::= ]
```

Test clauses may be nested to any depth providing the ability to develop a decision tree based on whether or not a particular test clause succeeded (S) or failed (F).

The setupProc is a Tcl procedure that sets up the parameters for the next outgoing RADIUS message. On return from the procedure, the test harness sends the message, puts the test instance in a wait list and continues processing other tests. When a response is received that matches the waiting test, the postProc is executed. This is another Tcl procedure that can examine the data returned from the RADIUS server. The postProc decides, based on the returned data, whether the step succeeded, failed or should be repeated.

The syntax describes a test object, which is the basic entity managed by the test executive. When a test is launched, a thread is created with an instance specific data area that continues to exist for the life of the test. This area maintains context and provides private data storage that may be used to communicate information from one step or procedure to another. At the completion of the test, all associated data storage is freed. The test harness allows instantiations of different tests as well as multiple instantiations of the same test to exist independently and simultaneously, each in their own thread. Communication between test threads is handled by global variables and default data values that may be defined for each sequence of tests.

As a concrete example, consider the following test template:

```
 1  session: send_auth auth_recv
 2  {
 3    F: log_bad_auth
 4    S: send_acct acct_recv
 5       {
 6       F: log_bad_acct
 7       S: send_acct_atop \
 8          [global max_session_length; \
 9          randno $max_session_length] \
10           acct_recv
11          {
12          F: log_bad_acct
13          }
14       }
15  }
```

Although shown as block structured, the parser is actually free format and all of the text might have been written on single line.

Line 1 contains the test name (session), the initial setupProc (send_auth) and the initial postProc (auth_recv). When the test is started, the send_auth procedure sets up the parameters for the initial RADIUS message. On return, the test executive sends the message and puts the test instance in a wait list. When an appropriate reply is received, the postProc auth_recv is executed. This procedure looks at the returned information and determines if the test succeeded or failed and returns an appropriate code. On a failure, the test executive evaluates the F branch on line 3 (log_bad_auth) and the test terminates. On a successful return, the send_acct setupProc on line 4 is executed followed by the acct_recv postProc when an appropriate reply message is received. Once again, the postProc decides if the communication exchange succeeded or failed and either the log_bad_acct procedure on line 6 is executed or the send_acct_stop and acct_recv procedures on lines 7 through 10 are evaluated. In the latter case, the Tcl expression enclosed in brackets is evaluated to generate an integer delay time. This is an optional value that may be used to add time delays in a test. When this value is not present, the test will execute each branch as rapidly as possible. In this instance, a procedure is being called to generate a random interval using a parameter set in a global variable, presumably set in supporting routines. On a successful return from acct_recv, the test terminates since there are no further test clauses.

This design provides a number of advantages. The overall test is described in a compact manner. Elements within a test description are Tcl procedures allowing almost unlimited flexibility and behavior options. At the same time, this allows for libraries of standard procedures to be constructed. New tests can then be constructed from basic building blocks.

Each test procedure is evaluated atomically by the test executive. Consequently, there are no concurrency issues when accessing shared data and variables. Test specific information is carried within an *instance data structure* which exists for the life of the thread and, in effect, provides an instance specific Tcl name space. When the test is started, the data area is created and a handle associated with the area is passed to all test procedures. A library of procedures based on the prototype implementation allows access to the data structure used to create the RADIUS message. Most test procedures consist of a number of calls to either set up the data structure or examine the returned data. The send_auth procedure, for example, might look like this:

```
proc send_auth {ds} {
#———————————————
# Set up the authentication request
global lastid user auth_port
global authtype rlm

# Get a user/password pair.
# Go round robin on it
incr user
if {$user > $lastid} {
    set user 1
    }

set usernm [format "%d%s" $user \
        [idSuffix $authtype]]

set userpw [makePasswd $usernm]

if {[string length $rlm] > 0} {
   set usernm [format "%s@%s" \
      $usernm $rlm]
   }

setRequestType $ds 1
setUser $ds $usernm
setPassword $ds $userpw
setPort $ds $auth_port
clearAvPairs $ds
setSeqNo $ds [ reserveSN ]
puts "Auth request for $usernm"
return
}
```

As can be seen, this is a standard Tcl procedure. The *ds* parameter is the handle to the instance data structure. The global variables are defined and initialized prior to the test execution. Test IDs used for authentication are generated on the fly in a round robin manner with the password generated from the ID by the *makePasswd* function. The bold text indicate functions which manipulate the RADIUS data structure.

3 Implementation

An overview of the architecture is shown in Figure 1.

A file containing the test template, possibly containing Tcl procedures used in the test, is compiled by a template parser. This is a Finite State Machine based parser which performs syntax checking and compiles the template into an internal tree format. Any Tcl procedures found in the file are added to the Tcl interpreter.

3.1 The Parser

The parser tokenizes the input making heavy use of the *regexp* function. The tokens are consumed by a Finite State Machine module comprising a scant 50 lines of Tcl code. In part, this compactness is due to the FSM description used. This is a Tcl list where each element is a tuple consisting of a state

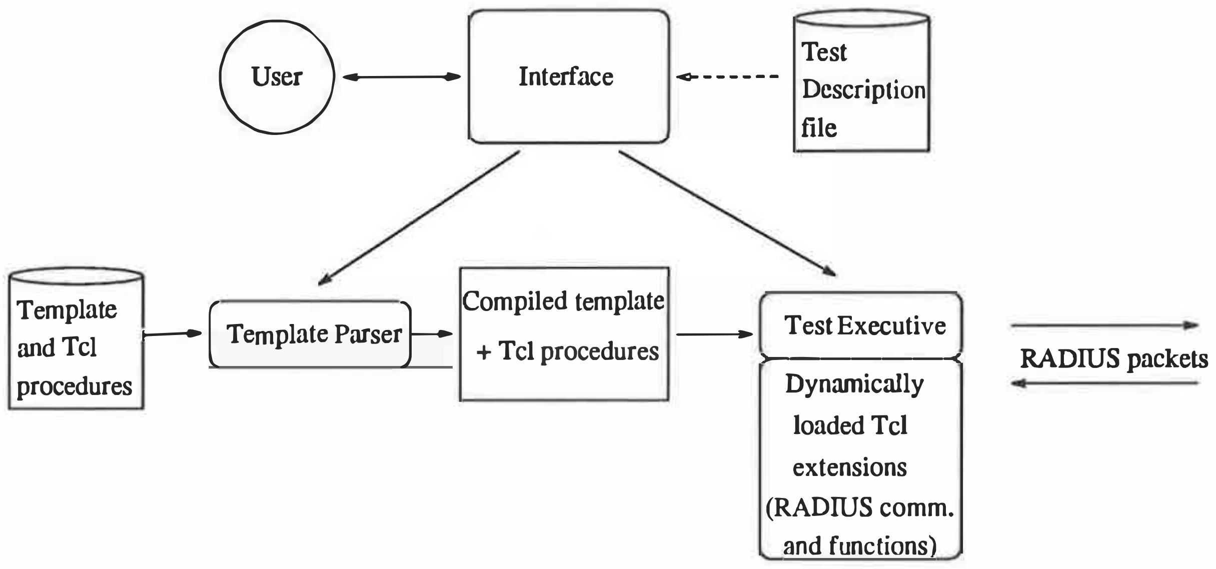

User
Interface
Test Description file
Template and Tcl procedures
Template Parser
Compiled template + Tcl procedures
Test Executive
Dynamically loaded Tcl extensions (RADIUS comm. and functions)
RADIUS packets

Figure 1: Architecture overview

name followed by one or more token/script/next-state tuples.

As an example:

```
{Start {{? {tclLabel} Start2}}}
{Start2 {
    {L {testLabel getnext} startTest}
    {X {} End}
    {null {} End}
    {? {{parseError "Test must start
        with a label."}} End}
    }
}
```

The state machine starts in the *Start* state and continues until it transitions to the *End* state. In the first rule any token matches the rule and the procedure tclLabel is called. This procedure checks for the reserved label *tcl_code:* which indicates that Tcl procedures following the label should be added to the interpreter. If found, this procedure sets the token type to X and the FSM will exit after it transitions to state Start2.

In state Start2, a label token (L) will cause the two procedures testLabel and getnext to be executed. The testLabel procedure sets up the environment for a new test while getnext causes the next token to be fetched. This rule will then transition to the startTest state. If an end of file condition was encountered, resulting in a null token, the FSM will transition to the End state. Any other token will result in the procedure parseError being called with the error message shown.

The parser uses the *eval* Tcl function to evaluate each procedure in the procedure list. Procedures requiring parameters, such as parseError above, are simply enclosed in curly braces so they become single elements in the list.

The code breakdown for the parser is as follows:

Function	Lines of Tcl
Tokenizing code	190
FSM code	102
State Machine description	61
Routines referenced by FSM description	242
Debugging routines	81

The line count includes comments and white space which inflates the line count 20-50 percent. The FSM code, for example, actually contains only 50 lines of active Tcl code.

Compiled test templates are stored in the Tcl global name space as an array called tst_*testname* with each test step stored as a numbered element of the array starting with array element 0.

3.2 The Test Executive

The test executive is at the heart of the test harness. It provides the basic functionality required to maintain multiple ongoing test threads. The executive is based on a simple Finite State Machine. When a test is started, the associated Tcl setup procedure is evaluated, the resulting RADIUS message is sent and the test goes into a Wait state. At the same time, an entry is place in a timeout list. If a response is not received by a settable timeout interval, the message is retransmitted and the retry count is decremented. If no response is received by the time the retry count reaches zero, or a total timeout interval has been exceeded, the test state will change to either R (retry) or X (exceeded) and the associated postProc will be executed. The values of the timeout interval and retry count may be controlled by the individual test. Following the postProc, the test will either end, if there are no more steps, continue processing with the next setupProc in the test, or delay some time interval before continuing with the next setup procedure. Figure 2 illustrates the FSM used for evaluating the tests.

Tests are loaded into the execution queue after they have been successfully compiled through a call which specifies the name of the test, the number of tests to run and an interval value used to space the tests. For added flexibility, the interval value may be either a fixed time interval, or a procedure which returns an interval value. While the interval may be specified to millisecond resolution, it determines the minimum time that will elapse between tests. The actual time will vary depending on the internal state of the test executive.

The test executive supports a number of additional functions to simplify writing test series. At the start of execution, prior to launching any tests, the test executive will execute the procedure *test_init* if it exists in the Tcl name space. Similarly, after all tests have completed, the procedure *test_end* will be executed. These procedures may be used to initialize global variables, handle logfiles and manage general housekeeping.

For each series of tests, a similar process occurs. In this case, the procedures are called *test_name_init* and *test_name_end* where *test_name* is the template label. These procedures are passed a handle to an instance data structure that will be used to initialize the instance data structures of every test in the series. Default initial conditions may be prepared or other tasks specific to the test series may be performed.

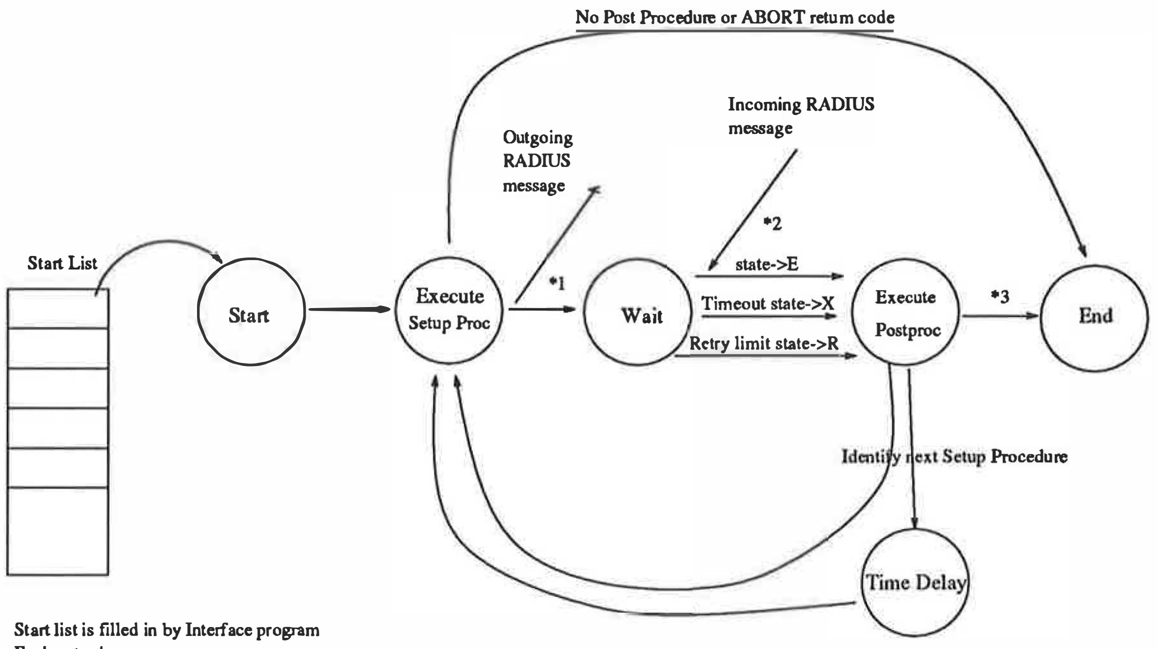
The test executive is written in Tcl with some supporting functions in C. The C functions provide for management of the instance data areas, access to the communications data structure used to formulate the RADIUS message and management of the communications functions, including matching incoming replies to the originating test. The management and execution of the test instance is handled by Tcl functions.

A peculiarity of the RADIUS protocol is that the only piece of information guaranteed to be present in the reply message that may be used to match incoming replies with outgoing requests is an 8 bit sequence number. This allows only 256 outstanding messages from a single source which would be a severe limitation for any type of load test. The current implementation solves this problem by using an array of sockets, any of which may have up to 256 outstanding requests. An array of 20 sockets has proven adequate in general use.

The initial implementation used a polling mechanism to manage the various state lists. Significant performance improvements were obtained by rewriting the C functions to use the dual-ported Tcl8.0 object interface. A final rewrite replaced the polling method with the Tcl event loop using timer (via the *after* function) and *vwait* events. Since the communication functions use C functions, the *fileevent* Tcl command was not applicable and an additional event type was required to indicate when incoming data became available.

The test executive totals approximately 480 lines of active Tcl code, excluding white space and comments. The supporting C code totals approximately 4000 lines. In addition, the test executive leverages the use of 15,000 lines of C from the Merit AAA Server code base to handle many underlying requirements when dealing with the RADIUS protocol.

As a side note, although only the RADIUS communications protocol has been referenced, the



Start list is filled in by Interface program
Each entry has:
- Name of test
- Number of times to run test
- Time interval between tests
- Time next test can be started

*1 When sending an outgoing RADIUS message, the test adds an entry to the outstanding request list. The test stays in the WAIT state until a response is received that matches the entry in the outstanding request list, the retry limit is exceeded or the total timeout limit is exceeded.

*2 Incoming RADIUS messages are matched to tests using the sequence number. The state is changed to EXECUTE and the postproc is executed with a status message of 'normal'.

*3 When there are no further steps in the test, the state changes to END and the test is finished.

Figure 2: Test Executive State Machine

test executive was written to allow the addition of other communication modules with minimal changes. Test scenarios were envisioned where out of band communication with some aspect of the server environment would be necessary, mandating this ability.

3.3 The User Interface

The template parser and the test executive combine to form a powerful test driver with a great deal of flexibility. The user interface serves to glue these pieces into a usable tool and focus the test onto specific tasks. To date, four different interfaces have been constructed.

- A generic test interface

This interface is a general purpose tool that is used for executing specific template files. The interface supports a number of standard command line arguments as well as a mechanism for passing arbitrary command line strings to the Tcl procedures in the template file.

- A session simulator

This interface simulates remote user login sessions sending authentication and accounting requests. Command line arguments allow session duration and generation rate to be specified. This is used to exercise the Merit AAA Server as well as to perform load testing.

- A file driven session simulator

This interface is similar to the session simulator, but the sessions are defined by a file containing timing and user information and possibly optional parameters for each session. This can be used to repeatedly generate a known user load with a specific makeup.

- A parameter driven test generator

This interface reads a test description file that describes each interaction with the remote RADIUS server. A default mechanism allows a basic RADIUS message to be defined that may be altered and customized by each following test clause. These customization strings are syntax checked and converted into Tcl commands stored in the instance data area. The template Tcl procedures use the eval function to execute these commands customizing the specific test instance. This interface has been used for regression testing and benchmarking.

All interface programs are roughly similar, share many of the same functions and are all approximately 300 lines of Tcl. The parameter driven test interface also includes another 200 lines of Tcl in support routines, primarily for parsing the test description file.

4 Performance

Initial development work for the test harness was done in Tcl7.6 running on a Sun Ultra-1/140. Under this environment, parsing a typical test template file required 1-2 seconds.

In order to better gauge test throughput, a test template was constructed that would immediately fail after sending a RADIUS request. This would cause the post procedure to be scheduled for execution immediately after the setup procedure completed. The setup and post procedures contained only a few Tcl commands necessary to gather statistical information, minimizing the time required to execute these procedures. This template provided an upper limit of the test throughput.

Using the initial polling implementation and Tcl7.6, the development system achieved a sustained throughput of 19 test steps per second. Although adequate for functional testing, this is far below the level required for load testing. Although not stated during the design stage, a goal of 1000 transactions/second was targeted as the utility of the test harness as a loading tool became apparent with use.

Using Tcl-8.0 with no other changes increased the throughput to 26 per second. Rewriting the internal C functions to use the Tcl 8.0 dual ported object interface and modifying some of the internal data structures to decrease search times increased the test throughput to 96 test steps per second.

Finally, rewriting the test executive to employ the Tcl event loop and removal of the polling loops allowed the development system to reach 149 test steps per second. Running this version on a DEC 500MHz Alpha achieved 340 test steps per second. This throughput is sufficient to meet the goal of 1000 steps per second if run as multiple processes on a multi-processor system.

5 Conclusions

Most of the goals set at the start of the project have been reached. The test harness has been in use for new release testing and benchmarking purposes. Canned tests using the defined interfaces are simple and convenient to use for the programming staff. The use of Tcl has allowed for a highly flexible test environment, rapid prototyping and development and the ability to leverage a significant body of existing code into the system. Development of key parts of the system often required only a few days due to the powerful nature of some of the Tcl primitives combined with the interpretive environment. Rewrite of the system from polling to event driven, for example, required only two days.

When running, the test harness typically uses 85-95 percent of the available CPU resources. In CPU intensive applications such as this, attention to execution details is very important. As an example, performance gains of 20 percent were observed when list structures were modified to minimize searching.

Performance is more than adequate for the majority of testing. The goal of 1000 transactions per second appears to be reachable using an appropriate multiprocessor hardware platform. Even the 130-140 transactions per second available in the development environment is sufficient to overload a targeted server on many of the commonly available hardware test platforms in use at Merit.

Porting to the DEC system mentioned in the Performance section was trivial requiring only recompilation on the target system. The test harness has been ported and run on SunOS, Solaris, DEC Unix, Linux and BSDi with no significant changes.

The most difficult area has proven to be the development of an interface and template combination that balances ease of use and flexibility. Members of the programming staff are not yet confident of their ability to develop new tests at this juncture. This area should improve over time as experience is gained in developing tests to exercise new features added to the software product.

6 Code Availability

Interested parties should contact John Vollbrecht at Merit Network, Inc. (jrv@merit.edu) for licensing details.

References

[Deutch] Michael Deutch, *Software Verification and Validation*, Prentice-Hall, Englewood Cliffs, NJ (1982).

[Libes] Don Libes, *Exploring Expect*, O'Reilly & Associates, Sebastopol, CA (1995).

[Myers] Glenford Myers, *The Art of Software Testing*, John Wiley & Sons, New York (1979).

[Savoye] Rob Savoye, The DejaGnu Testing Framework, `http://darkstar.cygnus.com/rob/dejagnu_toc.html`, (1996).

[Say] Janche Say, Ke-Hsiung Chung, Vernon Rego, *A Simulation Testbed based on Lightweight Processes*, Software Practice and Experience, **24**(5) (May 1994), p. 485-506.

[Stocks] Phil Stocks, David Carrington, *A Framework for Specification Based Testing*, IEEE Transactions on Software Engineering, **22**(11) (November 1996), p. 777-793.

[Tanenbaum] Andrew Tanenbaum, *Operating Systems, Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ (1987).

Using TCL/TK for an Automatic Test Engine

C. Allen Flick
DSC Communications Corporation
aflick@wss.dsccc.com, http://www.dsccc.com
James S. Dixson
Silicon Valley Networks Corporation
jdixson@svnetworks.com, http://www.svnetworks.com

Abstract

Test Automation Programming Interface with Object Controlled Authentication, TAPIoca, is a Tcl/Tk based automatic test environment. It grew out of the need to construct automatic tests at the system level that could send commands to a System Under Test (SUT), get responses from the SUT, and control signal generation equipment connected to the same SUT. Tcl/Tk was chosen as the basis for this automatic test environment because of the wealth and maturity of its "language" features, as well as its ease of extensibility.

1 Introduction

Automatic testing has been a widely discussed topic and desired goal for over twenty years in the electronics and software industries. Despite this, there is not one test engine, test management tool, or even test standard that meets all the needs, or desires, of every organization that might use it.

There is a plethora of reasons behind this phenomenon, not the least of which is the necessity for rapid turn around in authoring test cases and collecting their associated metrics to satisfy the needs of managerial reporting. This obligation was the main reason that drove us to choose Tcl/Tk as the language on which to base our TAPIoca automatic test environment. Tcl/Tk, and its extensions, gave us what we needed for the test engineers to generate well formed test cases.

The test cases written in the TAPIoca system could be assembled into scripts that allowed the test organizations to track the progress made in the development of the SUT. Using Tcl/Tk as the language basis of TAPIoca made it easy to assemble scripts and suites of scripts at a pace which could track the development of the SUT. This way tests could be ready for execution when the SUT was ready for testing. TAPIoca has also allowed our organization to quickly change test scripts, if the requirements of the product change (as they quite often do).

TAPIoca is still evolving. Our test engineers who use the tool continue to dream up new features they would like incorporated into it. We've always been advocates of the *creeping elegance* style of development, that is, software should be functional and usable before it can be feature rich. Our approach with TAPIoca has been that it should be used in *real* testing as early as possible in its development. This has allowed it to mature earliest in the areas which were most needed in our testing environment.

2 The Quest for the Ideal Tool

Fine cuisine for one is fodder for another. This is also true when trying to define an ideal test tool. Ask anyone with any significant experience in testing and test automation and you will get a long list of desired functionality ranging from the practical to the impossible.

2.1 A Reality Check

For some, automatic testing means only automatic "collecting" of test results, *i.e.*, a system that prompts the user to perform an action, then asks for the result. The automation system is nothing more than a data entry system for the results of a fundamentally manual activity.

For others, automatic testing means telling a piece of software "verify feature 'xyz' on test-bed '123"'. The test software proceeds to magically configure the '123' device, find the requirements of feature 'xyz', construct and elaborate a set of commands and signals to send to '123', send them to '123', then, using some embedded artificial intelligence, produce a conclusion about the feature 'xyz'.

In truth, reality lies somewhere in between these two extremes.

2.2 Good Automated Testing

Automatic test should do a lot more than just ask questions of a tester, but it most certainly will not be able to infer *everything* about what to do to perform a test.

Good automated testing requires that the test tool/environment have many *domain specific* features above and beyond general capabilities. To us, this meant that the tool had to be able to communicate with:

- RS-232 Devices
- GPIB Instrumentation
- Simulated development devices

In addition, we wanted the tool to have several general qualities:

- A short learning curve.
- Ease of GUI generation.
- Device Control Extensibility.
- A *full featured* scripting language.
- A standard, structured output format for all tests.

2.3 An Existing Tool

We were fortunate not to be designing in a vacuum. We already had an in-house developed test tool called Autotest, written approximately 10 years ago to address domain specific requirements. Autotest included a rudimentary scripting language that testers used to program their tests. It had many of the above mentioned general capabilities, and, overall, most were happy with Autotest, but, it had serious drawbacks:

- It ran only on MS-DOS.
- Its' scripting language was not *full featured.*
- Its' code required high maintenance.
- Its' extensibility was difficult
- It lacked a "batch" execution mode.

Much of our early effort focused on porting Autotest to a Unix environment. But, we realized that to do it right we would have to formalize the Autotest language. This was determined to be a significant effort. So much so, that it opened us to the possibility of using other tools or languages instead.

2.4 Commercial Tool Search

At the time Autotest was written there were no commercial products which could fit the bill. Ten years later when rewriting/replacing Autotest became imperative, we were disappointed to learn that little had changed.

The only two significant commercial contenders for the honor of *one true test tool* were LabView/LabWindows from National Instruments and HP-VEE from Hewlett-Packard. Both of them communicate quite well with GPIB-based test boxes and are relatively easy to learn. Several of our colleagues had developed some quite good, though very specific, automated tests in these environments. If these tools could be proven to do the more general domain testing that Autotest could do, then our quest would be over.

However, trials of both of these tools revealed a great deal of work would be required to implement communication with our SUT's. Furthermore, we found that while the graphical programming environments of LabView and HP-VEE are easy and fun to learn, when we tried to implement a representative sample of automated tests, the graphical programming environment became a real burden.

Using the LabWindows 'C' environment was even worse. It would require the testers to code everything in C. While it was certainly possible to invent a set of LabWindows C-API functions to make test authoring easier, this would have been just as much an effort as porting the Autotest communication code (also in 'C') to Unix.

2.5 Our Dilemma

At this time pressure was building to produce something that would be useful in doing real testing, and real soon. Our evaluation concluded that what our testers really wanted was a scripting environment like our internal Autotest tool, but with modern language features and capabilities.

Again, rewriting Autotest would have required far more time than was practical. Extending environments like LabWindows to include send/expect features *and* be easy to use was just as involved.

3 Our Big Discovery

Our first attempt at a redesign was through Perl, because of previous experience. So, Internet newsgroups were searched for extensions, or methods of using Perl, to meet our requirement of send/expect communication. Noting comments therein to Tcl/Tk and Expect, we looked into their newsgroups. There we found postings from some people right in our own backyard, DSC Communications. Then, further investigation into Tcl/Tk and the Expect extension showed us we had found the "treasure" we were hoping for.

Now that we had discovered Tcl/Tk and Expect, the solution to our dilemma was obvious. **Tcl** contained all of the fundamental language features that we were looking for. **Tk** provided us with easy GUI building that we felt we would need. And, furthermore, the **Expect** extension could easily do the send/expect pattern matching that we had implemented in the Autotest tool.

We also realized that we could easily create some wrapper functions around the Expect calls and end up with a language syntax that was not too different from the syntax of Autotest. This was a big win. This meant that both learning the new environment and converting legacy Autotest code to the Expect environment would be very straightforward.

4 The Prototype

The prototype of this Expect-based testing tool was coded and dubbed **TAPI**, for Test Application Programming Interface. It was a command-line driven application which accepted little more than the name of the script to be run and a few options.

It took little more than two weeks to code and test all the included functionality of Autotest, except GPIB instrument control. It became immediately obvious that **TAPI** had inherited some capabilities from Expect that made it far more useful than Autotest could have been. In particular, it was now trivial for a test script to be directed to a *simulated* SUT, rather than a real device. This would allow our testers to develop and debug tests without the need for time on expensive hardware.

TAPI also differed from Autotest in its Object-Oriented approach to devices under test. One of the things which made it so easy to develop tests within Autotest was that a tester simply had to identify a SUT by the port to which it was connected and send messages to it. Responses were automatically captured, formatted, logged, and compared to specified patterns. If the patterns did not match, an error condition was automatically logged. The tester had to do very little overhead coding to communicate with different SUT's, or log results in a standard format. We decided to adapt this model into a fully OO view of devices under test for **TAPI**.

Despite all these advantages, **TAPI** met with some resistance from the test group. The principle complaint was that while the language was far more capable than Autotest, the command-line nature of **TAPI** was intimidating. The Autotest application had a nice menu-driven front-end which made it easy to start/stop tests and monitor output. **TAPI** did not have that easy interface and, therefore, was not as *good* as Autotest despite the improved functionality. Appearances can be everything.

4.1 TAPI gets GUI

Using Tk, we were quickly able to implement an Autotest look alike interface for **TAPI**. The coding of the basic interface took about a week and had most of the same features of the Autotest interface. Around the same time, it also became known to us that a certain Redmond, WA, based software company had copyrighted the name **TAPI**, so a new name was in order. We decided on the name TAPIoca , which meant internally to us "a Gooey (GUI) version of TAPI", but officially came to be known as "Test Application Programming Interface with Object Controlled Authentication", to reflect the OO nature of the test scripting extensions.

TAPIoca was, and still is, a big hit. It is being used extensively by both our original test group as well as several other organizations in the company. The OO model for devices under test has allowed it to be used successfully in software development groups as well as test organizations.

4.2 GPIB Extension

We also implemented an extension to the basic Tcl interpreter to support communication with GPIB instruments. This turned out to be much simpler than we expected.

The extension capabilities of Tcl really shined here. All we did was create a set of "wrapper-functions" around the desired GPIB calls in the NI-488.2M library and link them to the interpreter.

Once it was done, it was *done*. Now, it's just another set of primitive functions that we use to construct higher level operations.

5 The TAPIoca Architecture

Like Caesar's Gaul, the architecture for the TAPIoca system is composed of three parts:

- TAPIoca API Function Set
- A set of standard **Test Objects**
- A Graphical User Interface

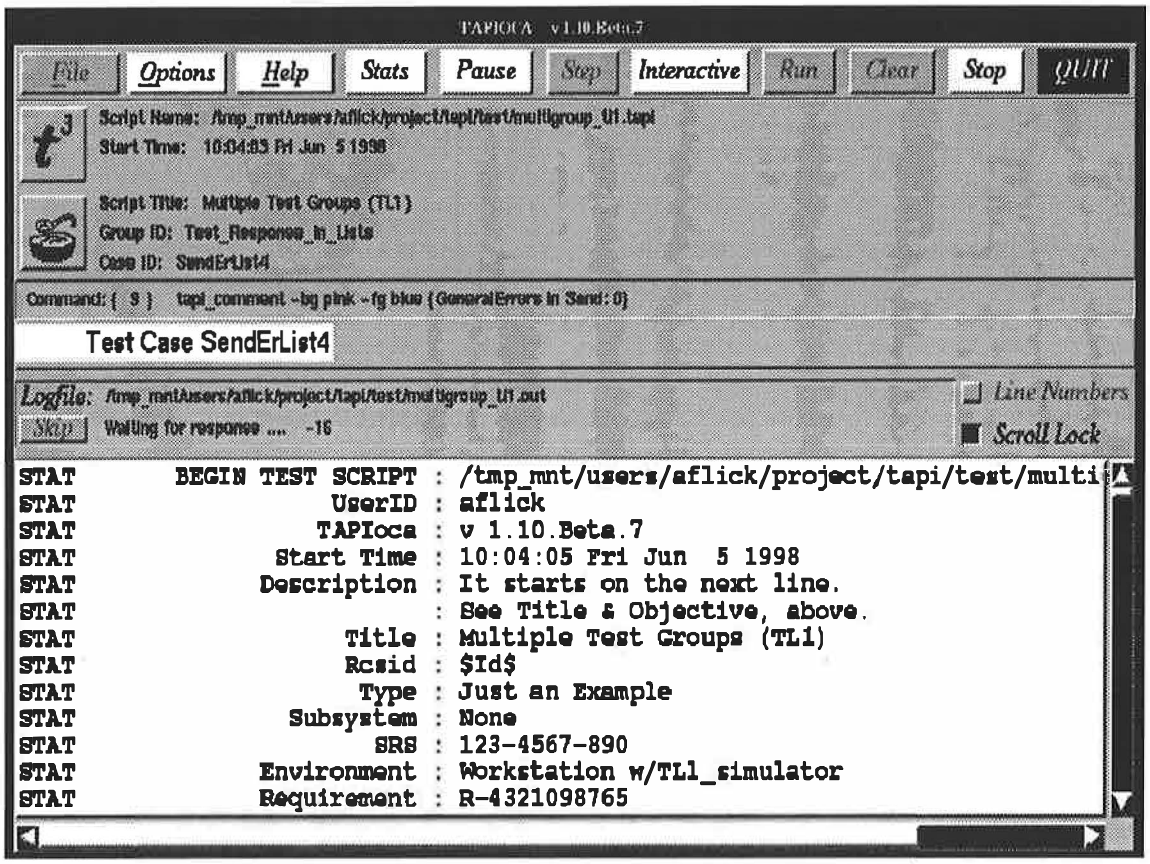
5.1 The TAPIoca API

The TAPIoca API, simply stated, is a set of procedures that implement functions in TAPIoca that are used within all test scripts. These functions provide the testers with the test structure, and overhead, that manage the following:

- Test Group blocking & metric collection.
- Test Case blocking & metric collection.
- Setting Test Group/Case Results
- Adding User Defined Statistics
- Changing Execution Options
- Changing GUI Options
- Creating a Pseudo-Random Number
- User Defined Comments to Output Log
- Displaying User Info to GUI
- Pausing for User Defined Time

Now, some of these, like the *User Defined Time* may appear redundant with core Tcl commands, but in our test environment these are simple extensions that do a little more than their Tcl counterpart.

Take the *User Defined Time* again. Very similar to the Tcl **after** command, but we needed to add to it a display to the GUI that shows the "count down" of the wait time. This is a legacy feature from Autotest that lets the user know the system is NOT dead, but it's actually waiting for something. A similar "count down" is used after a command is sent to the SUT while TAPIoca awaits a response from the SUT. Again, appearances can be everything.



Script Name: /tmp_mnt/users/aflick/project/tapi/test/multigroup_tl1.tapi
Start Time: 10:04:05 Fri Jun 5 1998

Script Title: Multiple Test Groups (TL1)
Group ID: Test_Response_in_Lists
Case ID: SendErList4

Command: { 3 } tapi_comment -bg pink -fg blue {GeneralErrors in Send: 0}

Test Case SendErList4

Logfile: /tmp_mnt/users/aflick/project/tapi/test/multigroup_tl1.out
Waiting for response .... -16

```
STAT       BEGIN TEST SCRIPT : /tmp_mnt/users/aflick/project/tapi/test/multi
STAT                  UserID : aflick
STAT                 TAPIoca : v 1.10.Beta.7
STAT              Start Time : 10:04:05 Fri Jun  5 1998
STAT             Description : It starts on the next line.
STAT                         : See Title & Objective, above.
STAT                   Title : Multiple Test Groups (TL1)
STAT                   Rcsid : $Id$
STAT                    Type : Just an Example
STAT               Subsystem : None
STAT                     SRS : 123-4567-890
STAT             Environment : Workstation w/TL1_simulator
STAT             Requirement : R-4321098765
```

Figure 1: TAPIoca GUI

5.1.1 API Structure

There is a standard format for all our TAPIoca API and internal **private** procedures. This was determined to be the best approach because testers, in their test scripts, can also "extend" Tcl by writing any procedures they may need. So, we came up with a standardized format and naming convention with which our testers were familiar.

The basic format of our procedures is as follows:

```
tapi_<command> ?args?
        or
tprv_<command> ?args?
```

Therefore, within our environment, any procedure starting "tapi_" is a TAPIoca API procedure, and any procedure starting "tprv_" comes from within the TAPIoca core. Testers are forbidden, by our code inspection process, to create procedures that begin with either of these prefixes.

5.2 A Graphical Interface

To appease our former users of Autotest and to create a test tool that would appeal to most new users, we created a Graphical User Interface.

One of our "unpublished" requirements was that we wanted all of the test scripts written in our replacement system for Autotest to be able to be executed in a "batch", or "headless", environment.

A "batch" mode implicitly means that the test environment cannot require a GUI for normal operation. This goal was foremost our minds when we began designing an architecture for the GUI. For TAPIoca we decided to literally "wrap" the GUI code around the **TAPI** core.

We wanted to preserve a "batch" mode in order to support scheduled execution of test on a remote platform. Previous experience has taught us it is much more difficult to schedule an automated test on a remote platform if that test requires a display to print information on.

We utilized Tk to create the GUI shown in Figure 1. We have our control bar across the top where our menu buttons and tool buttons reside. As shown, the state of each button is controlled during execution of a test script. The lower half of the GUI is what we call our "logview" window. This window displays, in user definable fonts and colors, the output of the test script's execution just as it is written into an output log file. We let the users define the colors and fonts therein simply as a method of making the GUI more acceptable to a wide range of users. Did we mention that "Appearances are everything"?

Controlling what is written into the logview is what we call our *File Viewer*. The *File Viewer* also allows the user to bring up a previously created log file and view it with the user's defined colors and fonts. The *File Viewer* also serves to limit the amount of memory that the logview consumes. We had a problem with limited memory when executing test scripts that generated extremely large output logs. So, by limiting the number of "lines" kept in memory, we minimize the problem for software, and memory consumption becomes a system usage problem.

The remainder of the upper portion of the GUI displays pertinent information for the tester. A quick glance will tell if the test script is operating correctly, and where execution is within the test script.

As shown, we list the active Test Script, Test Group, and Test Case. We also display the current TAPIoca command being executed and its relative position within the test script.

The design was to make the GUI as comprehensive as possible, but at the same time keep it simple and uncluttered. There is always somebody who wants "one more counter" or a "new button" to perform some very application-specific function. We fight vigorously against anything which involves GUI-specific functionality in the **TAPI** core. It is very important to us to defend the "headless" abilities of TAPIoca to run without a GUI interface.

5.3 The Test Objects

The "real guts" of what makes TAPIoca into a valuable test tool is what we call **Test Objects**.

Originally written before [incr Tcl] was available to us, we imitated the OO paradigm in TAPIoca by utilizing some of the "magic" features of Tcl. One of which is Tcl's native feature of allowing a script to define a procedure "on the fly".

To understand what we did, take a close look at the following TAPIoca command:

```
tapi_new <type> <name> <method> <entity>
```

The components of the `tapi_new` command are as follows:

type Declares this object to be a legal TAPIoca type.

name A user defined name used to refer to this object throughout a test script.

method The action to perform on the <entity>. It must match the associated kind of <entity>.

- **-open** Indicates the following <entity> is a *device* that needs "opening".
- **-cmd** Indicates the following <entity> is a program that needs execution.

entity The name of the *device* to be opened, or the program to be executed.

Shown below is a more realistic looking `tapi_new` command that shows the script defining a `GPIB` type object, to be referred to as `hp4532` throughout the test script, and we want device number 8 opened for the interface. In the context of GPIB, this device number refers to the **listen address** of the instrument to which communication is required.

```
tapi_new gpib hp4532 -open /dev8
```

Following a definition of this type in a test script, a procedure has been created by the name of `hp4532`, in our example. Later in the test script, communication with this device is then done by the command

hp4532, followed by one of various "methods" that is appropriate with this **<type>** of object. Such as issuing the command:

```
hp4532 -send "range7" ?expected-response?
```

If the **?expected-response?** is given, the actual response from the instrument is compared with the **?expected-response?** and flags are set accordingly, enabling the user's script to determine future action, if any, based upon the comparison's result.

There are several other common *methods* that each object type contains. And, some of our object types have **methods** that are specific to that type. Examples of other **methods** are:

-read Just read the device, or instrument.

-config Configure the device parameters (*e.g.* baud rate, parity, etc.).

-login Login to the device under test.

6 TAPIoca at work

There are many interesting things that, internally, TAPIoca does that we could emphasize, but, to be brief, we'll only look at a couple of them.

6.1 TAPIoca scripts

Figure 2 illustrates a typical test configuration. A workstation on the lab network is running TAPIoca, and is connected to the SUT via a RS-232 interface. This workstation is also connected to test equipment via a GPIB bus. The GPIB bus could be either an Sbus interface card internal to the workstation, as shown here, or a bus controller tied directly to the network with its own IP address.

In a TAPIoca test script, the tester could initialize the SUT object and the two GPIB test box objects as follows:

```
tapi_new tl1 msa -open /dev/ttya
tapi_new gpib hp4532a -open /dev6
tapi_new gpib hp4532b -open /dev8
```

Then, in this scenario, SUT setup commands could be issued via the "msa" object command. Likewise, commands to the GPIB test boxes could be accomplished via the "hp4532a" and "hp4532b" object commands. So, a typical test case scenario for a setup like this might follow these steps:

1. Issue SUT setup commands via "msa"
2. Issue GPIB setup commands via "hp4532a"
3. Issue GPIB setup commands via "hp4532b"
4. Read SUT status via "msa"
5. Change SUT setup via "msa"
6. Read SUT status via "msa"
7. Change GPIB setup via "hp4532a"
8. Read SUT status via "msa"
9. Change GPIB setup via "hp4532b"
10. Read SUT status via "msa"

A snippet of actual test code implementing a simple test case is given in Figure 3.

6.2 Test Result Logs

The bulk of output logs generated by TAPIoca are created by the Test Objects themselves. Therefore, when implementing a new Test Object in TAPIoca a great deal of thought is given to the output to be generated.

Each Test Object is responsible for logging all of its relevant activity to the TAPIoca log file. This way testers are not burdened with reporting the details as part of their scripting efforts. We wanted to make the test coding effort as "test logic" oriented as possible, off loading as much overhead and reporting tasks onto the TAPIoca system.

This approach in requiring Test Objects and the TAPIoca API to generate the log output rather than the test scripts has its greatest benefit in ensuring conformity. No matter how poorly coded the test

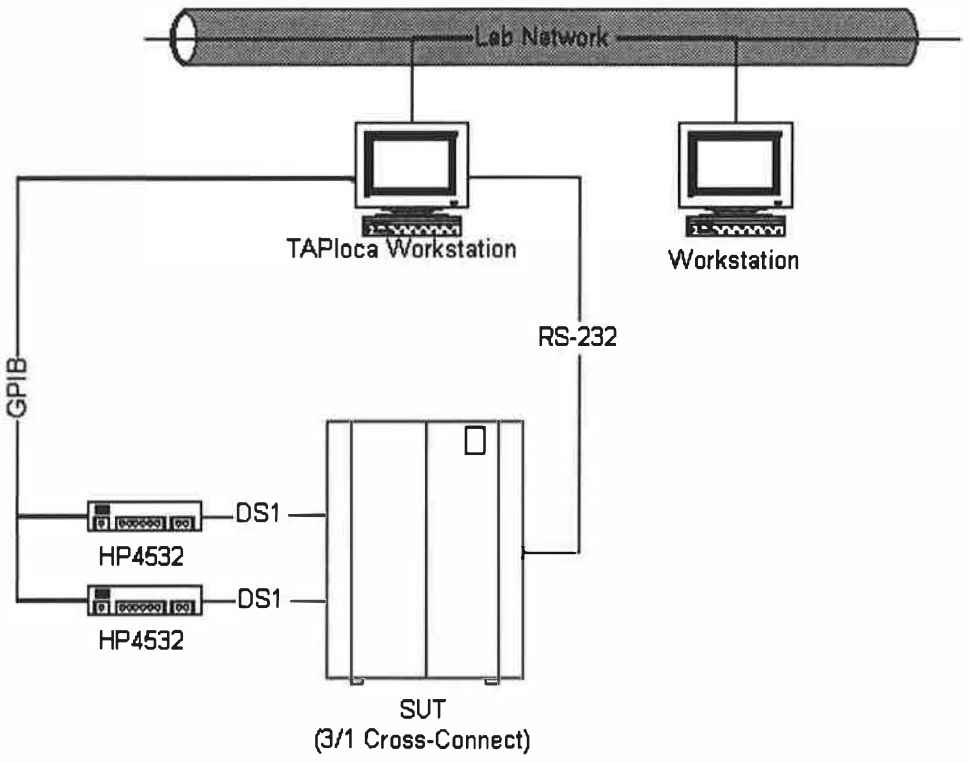


Figure 2: Typical Lab Network Configuration

script is, the output log generated will always contain a complete record of the steps applied to the SUT. This abstraction also allows the output format to change globally without having to change the logic in test scripts.

Figure 4 shows the output generated from the snippet of test code given in Figure 3.

6.3 Integration with Other Tools

The OO abstraction provided by the TAPIoca architecture has permitted TAPIoca to be enhanced to take advantage of newer technologies available. One of the most important of these is TAPIoca's integration with the TestExpert test management system.

One of the features of the TAPIoca environment that was inherited from the desire to work like the Autotest tool is generating a great deal of structured output automatically. The reason for this in Autotest was to support the parsing that output and extracting test metrics automatically. This was never realized in the Autotest world because maintenance of the tool was so high that test tool development had no time for new projects.

Much of the time spent maintaining Autotest internals was freed up after the adoption of TAPIoca. This allowed test tool development to focus on other problems like the management of tests and the collection of metrics.

We decided that the best test management tool would be one which would automatically parse/distill the results of TAPIoca test and insert them into a SQL database. Reports could then be generated from that database.

We were all ready to start development of this system when we discovered the commercial tool TestExpert. TestExpert is a test management system from Silicon Valley Networks that did just what we wanted. It would execute your tests and import the results into a SQL database. It even included a set of canned reports to get us started with report generation. The only problem with TestExpert was that it would only import files in the Test Environment Toolkit journal format into its database.

```
tapi_test_case Number213 {
    tapi_comment "Connect testport 288 to monitor mode"

    msa -send "ttst mon to 0108024 tp 288" \
"M ..:..:.. ..,.. . TTST MON .......,0108024 TP 288 2 LN MSG:<cr><lf>
TRSP TRB SG T CGA.O. TLA R COMPL<cr><lf>"

    tapi_fail_case -miscompare msa "Testport 288 unable to go to monitor mode"
}
```

Figure 3: TAPIoca Code

After some investigation, we realized that the TET format was not much different from the structured output that TAPIoca generated. Taking advantage of our OO architecture, as well as the TET API code provided with the TestExpert product, we were able to get TAPIoca to generate TET journal output natively without any changes to the test scripts themselves.

We can now run TAPIoca test scripts under the control of TestExpert and take advantage of TestExpert features like remote and scheduled execution.

7 The Future

The TAPIoca architecture is based upon our own internally developed OO support code written in Tcl. Going forward, we want to transition this code to the OO model of [**incr Tcl**].

Some work in this area has already been done, but more needs to be accomplished, and we must be careful in doing so, to keep existing test scripts executable.

This fact, as well as the lack of a *released* [**incr Tcl**] for the latest version of Tcl/Tk (8.0), is keeping us from upgrading to 8.0, but we plan to do so as soon as possible.

We also would like to port TAPIoca to the Windows NT environment. One of the requirements of our industry often involves the testing and certification of devices at the customer site. By porting to the NT environment we could more readily take advantage of the lower cost and higher availability of Windows-based laptops.

Last, but not least, is evangelism. We continue to promote TAPIoca internally. Several groups outside our product organization have adopted TAPIoca as their principle test environment. The extensible capabilities of both TAPIoca and Tcl/Tk has spawned several other test groups to write their own TAPIoca test objects.

8 Acknowledgments

When both of us were at DSC Communications, we worked in the same organization. Because of that, our acknowledgment list is a common one. We deeply convey a hearty *Thank You* to all those in this list, both for their role in development and support of TAPIoca within DSC Communications.

Walt Mamed : Our team leader who kept us on track when we wanted to pursue tangential paths of development.

Cheri Snapp : The original manager of our tools group who allowed us the latitude of development we needed, when we needed it. She has championed this test tool for some time and continues to do so in her current position in another division of DSC Communications.

Mark Scafidi : Test manager in another product line within DSC who showed great patience and trust in allowing us to demonstrate that TAPIoca could do **Tk GUI** testing where other products had not met his needs.

And, a special *Thank You* goes to **Jim Edmunds**, our Vice President of product development who provided the funding and significant moral support for

```
CO        =====================================================
STAT          BEGIN TEST CASE : Number213
STAT              Description : Setup odd split mode of fifth set of 12
                              : two-way mapped testports.
STAT               Start Time : 13:14:59 Fri Jun  5 1998
CO        -----------------------------------------------------
CO        Connect testport 288 to monitor mode
SND(msa)      ttst mon to 0108024 tp 288<cr>
RCV(msa),R  M 01:15:08 01,00 4 TTST MON 108024 TP 288 F FAIL DNY<cr><lf>
RCV(msa),R  <si>
ERROR     ******************************************************
ERROR     Miscompare Error
ERROR     Expected:
ERROR     M ..:..:.. ..,.. . TTST MON .......,0108024 TP 288 2 LN MSG:<cr><lf>
ERROR     TRSP TRB SG T CGA.0. TLA R COMPL<cr><lf>
ERROR     ******************************************************
RESULT      TEST CASE SUMMARY : Number213
STAT                   RESULT : FAIL
STAT                   REASON : Testport 288 unable to go to monitor mode
STAT                 End Time : 13:15:16 Fri Jun  5 1998
STAT              Miscompares : 1
STAT        END TEST CASE LOG : Number213
CO        =====================================================
```

Figure 4: TAPIoca Output

the project, helping us keep focused on the goal of creating a *practical* test automation tool.

9 Availability

Currently TAPIoca is an internal tool only available inside DSC Communications.

Based on the history of the *Mega Widgets* extension being associated with DSC Communications, we are negotiating with management to release the core of TAPIoca that would not be considered proprietary to DSC Communications. We hope to make that available by year's end.

References

[Expect] Don Libes, *Exploring Expect*, O'Reilly & Associates, Inc. (1995).

[Ousterhout] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishers (1994).

[NI488] *NI-488.2M Software Reference Manual, Sept 1994 Edition*, National Instruments Corporation (1994), Part Number 320351B-01. http://www.natinst.com

[TEAdmin] *TestExpert, The Test Management System: Administrator's Guide*, Silicon Valley Networks (1997), http://www.svnetworks.com

wshdbg - A Debugger for CGI Applications

Andrej Vckovski
Netcetera AG
vckovski@netcetera.ch

Abstract

This contribution discusses wshdbg, an interactive, remote debugger for CGI applications written in pure or derived Tcl-based environments such as websh. The discussion covers a short overview of the websh environment and an analysis of current techniques and impediments of CGI debugging. The debugger presented consists of a client-server architecture, where the server is running on the same host as the Web browser, while the client is included in the CGI application that needs to be debugged. The resulting environment does not provide the level of sophistication known from typical source level debuggers, yet it presents a significant step forward compared to typical CGI debugging techniques which rely on tracing the execution by many log or debug messages. The system has been in operational use in the last two years and proved to be a great help in debugging large Web-based applications, allowing high-level software engineering for Web applications on nearly the same level as it is being done in traditional software development.

1 Overview

This contribution presents an interactive debugger for CGI (Common gateway Interface [1]) applications written in pure Tcl or derivations thereof. CGI has been the first standardized technique that has been available to extend Web-servers to provide dynamic content and it is still - despite of some drawbacks - the most frequently used method to interface specific applications for dynamic content creation.

Debugging such CGI applications has always been a difficult and cumbersome task. CGI applications are growing in size and complexity and suitable debugging techniques need to applied to support the software engineering process. The lack of useful tools motivated the development of a simple yet powerful remote debugger for CGI applications which has been primarily targeted for the use in the websh-Framework which will be discussed below. However, the debugger can be used within other Tcl-based environments for Web-Applications such as Don Libe's cgi.tcl [2] or Neosoft's NeoWebScript [3].

The first section will give a short overview on the web++/websh-framework and motivate the use of CGI as server extension mechanism. This is followed by a short discussion of the generic difficulties in debugging CGI applications and a review of debugging techniques typically used. Then, the architecture and some implementations aspects as well as examples are discussed. The contribution is concluded by an outlook to our future work and some lessons learned using the debugger in various projects.

2 The web++/websh framework

The web++/websh framework is a software development and runtime environment for Web-based applications. It consists of two major components:

web++

web++ is a C++ class library which provides utility classes for many standard tasks such as managing the CGI protocol, logging, session management, (HTML) template processing and so on, much similar to the well-known packages such as CGI.pm [4] or cgi.tcl [2].

websh

websh (pronounced *web-shell*) is a Tcl-shell which is based on web++ and provides most of the functionality of web++ on a script level. The objects and methods within web++ are made visible on the Tcl-level as Tcl-commands.

When web++ was developed a few years ago the main incentive was to build Web-based applications as a specialization of the web++ class library, using the embedded Tcl interpreter merely as a very comfortable and powerful way for application configuration and template-based page construction. However, the experiences have shown, that a classical C++-development approach with relatively long turnaround cycles and high level of sophistication by the developers does not meet the requirements of typical web-based applications. Therefore, the entire functionality of the web++ class library has been exported as Tcl-commands in a specialized shell (websh) and the application development happens by writing websh- or Tcl-code, respectively.

Unlike usual CGI applications in Tcl, websh processes the code after evaluating the script file(s). The code merely declares callback procedures or code blocks which are called by websh after initializing and parsing the CGI input. This allows a very flexible yet simple way to write applications with *multiple states.* A command dispatcher selects the state requested by the corresponding URL (given by an encoding in the so-called *query string*) and calls (or evaluates) the appropriate callback code.

Other features of websh include:

Session management

websh implements a session management on top of the connectionless HTTP-protocol. Sessions can be associated with persistent *dictionary* objects (key-value pairs) on the Web server.

URL encryption

websh provides a simple URL encryption scheme to hide parameter passing details. This is necessary because session keys and state information are passed within the URL's query string.

Flexible logging

websh implements a two-level logging scheme. Every log message which is generated within an application or the interpreter itself consists of a *message text*, a *severity level* (debug, info, error, alert) and a free-string *facility code.* In the first stage, the severity level and facility code are used to filter the messages according to a user defined set of rules, e.g., "pass all messages with level info and debug messages for facility foo". A message that passes the filter is then routed to a set of log output channels. A log output channel can be either a file, any opened Tcl channel (e.g., pipe, socket, file) or, on Unix systems, the syslog service. Every log channel is also associated with a rule that defines which messages to accept. E.g., it is possible to route only alert messages to syslog and having all other messages in a log file. The logging mechanism is insofar very important as most Web-based applications are expected to run unattended without permanent operator control.

Template processing

The template processing used in websh allows templates (e.g., HTML page templates) to be processed with substitution of embedded directives. Templates can be entire HTML pages or only fragments thereof. The mechanism adopted is unlike most other approaches in that it uses a tagging which is *orthogonal* to SGML or HTML, respectively. This allows nesting within the templates and avoids possible name clashes with future HTML extensions.

Database connectivity

websh and web++ provide a lightweight database connectivity layer called webdb++. It allows simple (mostly synchronous) database connections to relational, object-relational and inverted-list systems. The database connectivity layer is implemented using database connectors that are available in the Tcl community such as Oratcl or Sybtcl [5].

More than CGI

The main usage of websh is to develop CGI applications. However, most large CGI application do contain components which need to run independent of a Web-server, e.g., housekeeping processes, import and export tools and so on. websh allows to reuse modules both within the CGI and non-CGI part of a software system.

Development tools

A set of development tools, such as a pseudo-linker (merely a module merger), a pseudo-compiler ("compiles" the websh/Tcl-code into shared objects), pretty printers and so on are available.

The powerful logging mechanism does provide a step towards *organized* and *controlled* CGI debugging. However, it is still far away from the usability that is known from traditional source level debuggers, e.g., debuggers for C/C++ applications. For that reason, we designed and developed a more powerful debugging approach which is somewhat close to the ease-of-debugging found in state-of-the-art source level debuggers. The next section discusses some of the difficulties that have to be overcome with a successful CGI debugging approach.

3 Debugging CGI-Applications

3.1 Why CGI?

Before discussing CGI debugging in more detail it is worth considering the question whether CGI is at all appropriate for the development of Web-based applications. In the last years many other approaches for dynamic content creation have been proposed and developed:

Server-specific APIs

Many HTTP-servers provide specific application programming interfaces (API) to extend their functionality and embed dynamic content creation, such as NSAPI (Netscape [6], ISAPI (Microsoft) [7], Apache modules [8] or *Servlets* for Java-based servers.

Server-side includes (SSI)

Some HTTP-servers provide a simple template processing mechanism. SSI has been available with very early releases of the NCSA-web-server. Microsoft IIS uses a similar approach called *active server pages*.

Approaches particular to the back-ends

Web-applications that need to access mainframes or large databases can use specific HTTP servers that are particular to the back-end system, e.g, Oracle Web Server [9].

Compared to these approaches CGI has a major drawback: It requires the HTTP server to create a process ("fork/exec") for every request that has to be handled. Process creation has been very expensive in older operating systems and is still expensive compared to, e.g., a function call or thread creation which is needed in the case of a server-specific API. That is, CGI is not an optimal solution if performance is the major issue. However, there are two fundamentally important reasons which make CGI still the technique of choice for Web-based applications:

- CGI application are isolated in a separate process space. Ill-behaving CGI applications cannot impair a HTTP server's stability (assuming a *real* operating system). For many applications, long-term stability is much more important than a few performance issues that can be solved by better and faster hardware.
- The CGI is well-defined and standardized and therefore, portable across many server platforms. Dependency on a specific HTTP server products limits scalability and flexibility.

These two issues have motivated us to stick with CGI for most of our Web-based applications. The development framework is, however, not *fundamentally* bound to CGI. In the next section we will now discuss some of the impediments when debugging CGI applications.

3.2 Impediments

CGI applications receive their input using two separate mechanisms:

- Context information such as server information and so on are made available by the server using a set of defined variable in the process' environment.
- Request-specific data generated by the browser such as the content of a HTML form are passed on the process' standard input channel.

The response is send by the CGI process to its standard output channel, which is either collected by the server and sent to the browser or, already connected to the browser socket by the server (buffered vs. direct replies). CGI applications usually have a very short life time. The process dies as soon as a request is handled (i.e., the response is sent to the standard output). A multi-state application (e.g., a shopping bag application) typically consists of many request/response pairs, i.e., of many calls to the CGI application. Every instance might expect some other data on the standard input and a different context in the environment.

Therefore, a realistic debugging session needs both the expected data on standard input and the environment variables to be set according to the request that needs to be handled. Assume a CGI application written in a 3rd-generation language such as C. Using standard debuggers for C there are basically two alternatives for debugging:

- The CGI is simulated using a few environment variables and some captured data which are fed into the process' standard input. The process is started manually or by the debugger, respectively, and not by the HTTP server. This approach is very cumbersome because it needs manual maintenance of the input data (which can be different with every request) while having very short debugging sessions by the nature of CGI processes [10].

- The CGI application is started as usual by the webserver. The application is extended in a way that it stops and waits for any signal to proceed (e.g., loops until some condition is true). During this wait period, a suitable debugger can attach to the process and force the condition to become true. There are also tools that can be used to connect to the debugger from the application (i.e., the other way round). The drawback of this approach is that the application needs to be significantly extended to allow such remote debugging. It is, however, still the best method to do it. However, it requires powerful debuggers which allow, e.g., to attach to a running process. Also, this approach requires in the most cases that the debugger runs on the same system as CGI application. On productive systems, this is most often inhibited due to security reasons.

For scripted CGI applications (Perl, Tcl, Python etc.) the technique used most often is still the classic "`printf()`" debugging style. Many log messages are generated in the process giving information about the state of variables and flow of control. The log messages are either sent to a specific log file, that process' standard error (which is often copied into the server's error log) or even sent to the CGI application's output, i.e., intermixed with the "real" output (which works only, if the output is ASCII-text or HTML and not some other, binary-encoded MIME-type such as GIF images).

Often used are also wrappers that are called instead of the CGI application. These wrappers call the CGI applications in turn and provide a formatted output of the application's standard input, environment and the results of the CGI application.

Compared to "real" debugging environments these techniques seem to be anachronisms and far from supporting productive software development. The `wshdbg` (`websh`-Debugger) presented in the next section tries to overcome some of these impediments and provide a better way for debugging Tcl-based CGI applications.

4 websh-Debugger

4.1 Architecture

The `websh`-Debugger `wshdbg` is designed as a *remote* debugger which consists of a client-part and a server-part (see figure 1). The server part is a `wish`-Application which displays the debugger's user interface and waits for CGI applications to connect. The client part is a small stub which is included in the CGI-application when in debugging mode. When a CGI application is launched by the HTTP-server, it connects to the debug server and transfers all context information (environment, decoded data from standard input, decoded query string). The debug servers typically runs on the same platform as the Web browser and controls the further execution of the CGI application. After the first connection to the server, the debug clients awaits further commands from the server (e.g., continue execution). The CGI application may contain a set of break points and trace conditions (variable traces). Whenever a breakpoint is reached or a trace condition is met, the execution is stopped and control is "transferred " back to the server. In a stopped state, the CGI application accepts various commands form the server. These commands can be, for example, a valid Tcl command that can be used to query variables, temporarily evaluate expressions and so on.

This design is similar to corresponding approaches for remote debugging known from standard debuggers available on most platforms. Remote debugging is especially useful if disturbing the debugged platform needs to be minimized, such as kernel programming or applications with high GUI requirements (e.g., the debugger GUI should not interfere with the debugee's user interface). However, to our knowledge, `wshdbg` is the first operationally used debugger for CGI applications which uses a remote debugging technique. Other debuggers have been successfully used in interactive Tcl/Tk environments such as for example [13]

Breakpoints and trace conditions are inserted in the debugged application - being a Tcl or websh-script - directly in the source code. This is a drawback which is imposed by the interpreted nature of the environment. Practical use has shown, however, that this does not pose major impediments.

The debug server can simultaneously control several CGI applications, i.e., maintain several connections to CGI applications. This is useful if there are, for exam-

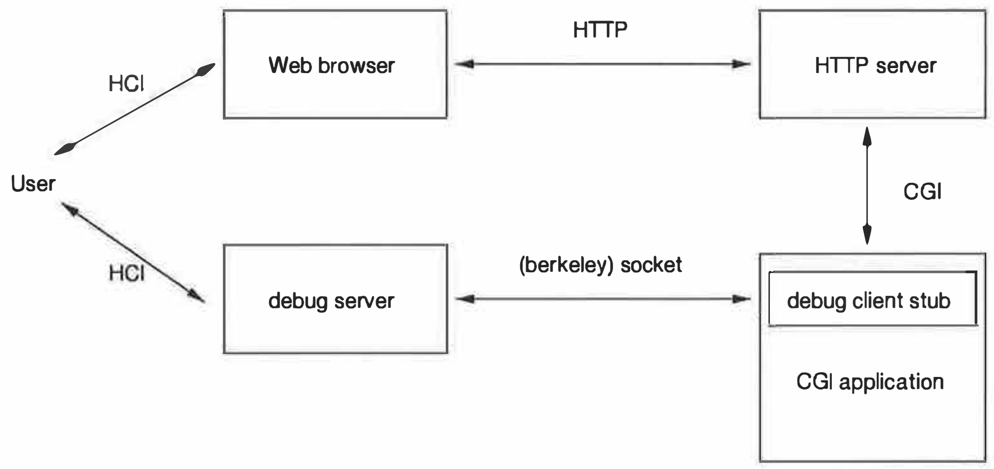


Figure 1: Architecture overview

ple, multiple requests needed for a response such as if both a HTML page and an embedded image are result of different instances of the same CGI application.

4.2 Implementation

The implementation is based on a simple bi-directional, message-oriented protocol. The protocol is implemented upon (berkeley) sockets and distinguishes two states, namely

1. the CGI application is stopped, and
2. the CGI application is running.

The *stopped* state is entered at start, at end (optional) and whenever a breakpoint is hit or a trace condition is met. The stopped state is left by selecting the *continue* action in the debug server. In a stopped state (i.e., the CGI application is awaiting commands from the debug server), the debugger provides following actions:

- The CGI environment (environment variables, decoded standard input etc.) can be inspected
- Tcl-code can be evaluated in the current context of the CGI application, allowing variables to be inspected and set, expressions evaluated and so on.
- Profiling can be enabled or disabled. The profiling can be used if the TclX [9] extension is available on the HTTP server platform . However, unlike typical profiling it collects profiling information over many instances of the CGI application, allowing better statistics for short-lived CGI applications.
- The CGI application can be prematurely terminated.
- The CGI application's execution can be continued. The CGI application continues to run until the next breakpoint is encountered or a trace condition is met.

On transition from a running state to a stopped state the client sends the server information about *where* it stopped. The location indication is a user-defined text which has been given when defining the breakpoint or trace condition.

Breakpoints and trace condition are defined as previously mentioned directly in the source code. A `brk` command (which takes optional arguments naming the location) defines a breakpoint. the `tr` *`variable`* command declares a trace condition which is implemented using the variable tracing mechanism of Tcl. In a typical debugging session, the source modules are kept open in an editor and break points are inserted as needed. This requires, however, that the code is debugged in an non-compiled or "obfuscated" state.

The following code snipplet shows the enabling of the debug client as well the definition of some breakpoints

```
wpp_command showdata {

    # use debugger (by the default,
    # connect to the address where
    # the request came from)
    usedbg

    set foo {some1 data1 some2 data2}

    foreach {key value} $foo {
brk having key $key
        wpp_puts "$key = $value"
    }
tr myvar "trace on myvar"
    # ... some processing

brk here is breakpoint 2

    # ... some more processing
}
```

Figure 2 shows a screen shot with the main debugger control panel, the CGI inspector and expression evaluator.

4.3 Log-Viewer

In the websh-environment, there is another useful tool available which supports debugging with additional information. The websh logviewer wshlv is a log server which runs similar to the debug server on the same host as the browser. When enabling the debug mode for the CGI application, a log channel to the log viewer (if available) is opened automatically. Every log message generated in the application and framework is sent to that log viewer. The log separates and stores log messages from various instances of the CGI application. Various runs can be compared without having a huge log file that contains intermixed messages from many requests. Figure 3 shows a sample screenshot from the log viewer.

Together with the log viewer, a typical debug session contains therefore of:

- a browser
- a debug server instance,
- an editor with the source modules, and
- a log viewer.

5 Conclusion and future work

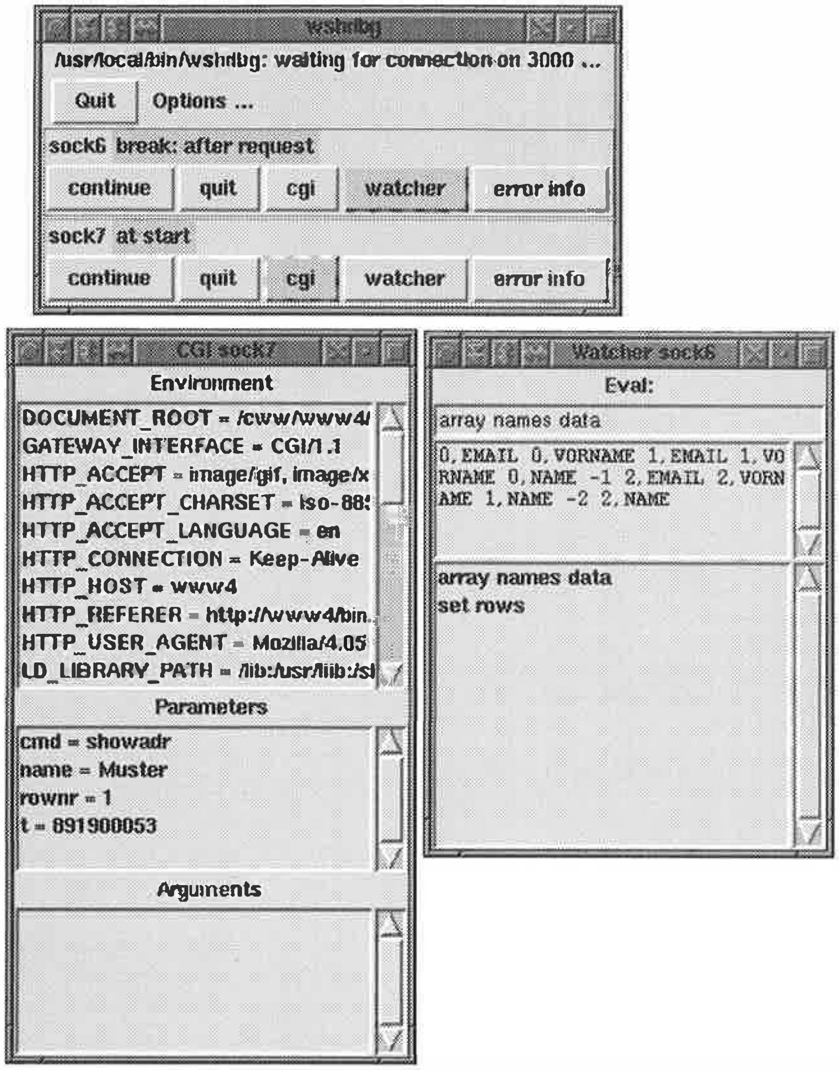
The websh-debugger wshdbg has been successfully deployed in many large projects during the last two years. We have experienced a substantial improvement in debugging productivity compared to classical `printf()` style of debugging. Especially, if the application is dependent on the real server (and browser) environment, this approach has an important advantage. Often, such information can not be easily simulated as it involves many other components (e.g., authentication information provided by authentication services, mainframe connections and so on). In these cases, it is very helpful to debug applications in their real framework, yet having full control and inspection at runtime of the application. Compared to these advantages, it is acceptable that the debugged application needs to have a few additions (such as the inclusion of the client stub).

It might seem unsatisfying that the definition of break points and trace conditions has to happen directly in the source code. Especially if the application consists of several source modules which are merged into a large, single executable, this involves a "make"-style step after every breakpoint has been included or removed. However, these steps are usually rather quick in script-based environments since a build of an application merely consists of the merging source modules (unless there are packaging concepts used - which is usually not acceptable in production environments).

The entire debugging system is in fact a very small piece of code - the debug server consists of less than 500 lines of Tcl-code, the client stub some 200 lines of Tcl-code. Yet, it provides most of the functionality needed for the debugging of large Web applications.

The future work on the websh-debugger could consist of extending the protocol between debug server and client to allow additional break conditions and a better user interface for inspection. However, it might be sensible as well to integrate the commercial debugger Tcl-Pro Debugger from Scriptics, Inc. [12], as it certainly provides much more general debugging functionality than wshdbg.

In general, we would like to have a source level debugger that allows break points to be set without modifying source code by specifying file name or procedure name and line number and still is suitable for CGI development. In an interpreted environment, however, the help of the Tcl interpreter is needed. An possible extension to wshdbg would be the possibility to use the Tcl



```
/usr/local/bin/wshdbg: waiting for connection on 3000 ...
sock6 break: after request
sock7 at start
```

```
DOCUMENT_ROOT = /cww/www4/
GATEWAY_INTERFACE = CGI/1.1
HTTP_ACCEPT = image/gif, image/x
HTTP_ACCEPT_CHARSET = iso-88
HTTP_ACCEPT_LANGUAGE = en
HTTP_CONNECTION = Keep-Alive
HTTP_HOST = www4
HTTP_REFERER = http://www4/bin.
HTTP_USER_AGENT = Mozilla/4.05
LD_LIBRARY_PATH = /lib:/usr/lib:/s
```

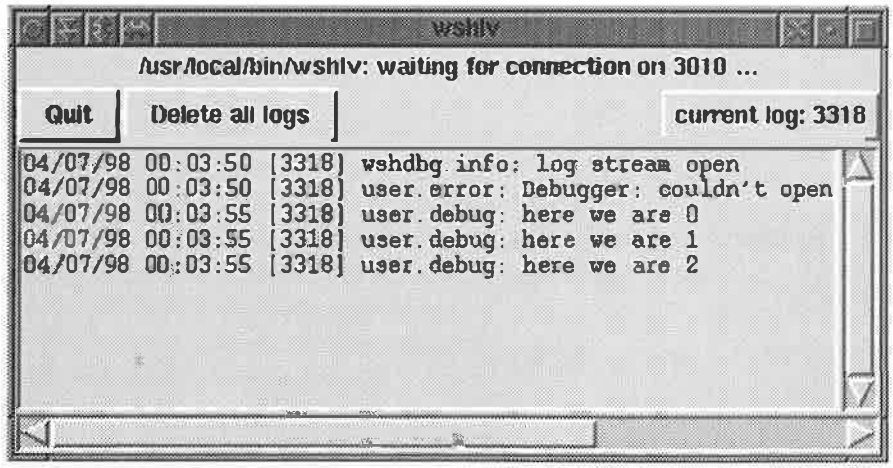
```
cmd = showadr
name = Muster
rownr = 1
t = 891900053
```

```
array names data
```

```
0,EMAIL 0,VORNAME 1,EMAIL 1,VO
RNAME 0,NAME -1 2,EMAIL 2,VORN
AME 1,NAME -2 2,NAME
```

```
array names data
set rows
```

Figure 2: Debugger screen example



```
/usr/local/bin/wshlv: waiting for connection on 3010 ...
current log: 3318
```

```
04/07/98 00:03:50 [3318] wshdbg.info: log stream open
04/07/98 00:03:50 [3318] user.error: Debugger: couldn't open
04/07/98 00:03:55 [3318] user.debug: here we are 0
04/07/98 00:03:55 [3318] user.debug: here we are 1
04/07/98 00:03:55 [3318] user.debug: here we are 2
```

Figure 3: Log viewer screen example

library function `Tcl_CreateTrace()` as it is used, e.g., in TclX [11] to provide call backs that test for break conditions. Very useful would be, however, to provide a mechanism within Tcl that provides additional information such as the line number within the upper stack frame that executed the corresponding command. This would allow the debugger to recognize break conditions on module and line level. However, this would require the Tcl byte code compiler to operate in a specific debug mode where line number information is not only retained in the case of an error.

6 References

1. *The Common Gateway Interface Specification* `<http://hoohoo.ncsa.uiuc.edu/cgi/interface.html>`
2. *The cgi.tcl Home Page* `<http://expect.nist.gov/cgi.tcl>`
3. *NeoWebScript* `<http://www.neosoft.com/neowebscript/>`
4. *CGI.pm - a Perl5 CGI Library* `<http://www-genome.wi.mit.edu/ftp/pub/software/WWW/>`
5. Harrison, M. (ed.), 1997, *Tcl/Tk Tools*, O'Reilly and Associates, Cambridge (MA).
6. *NSAPI Programmer's Guide* `<http://developer.netscape.com:80/docs/manuals/enterprise/nsapi/contents.htm>`
7. *Taking the Splash, Diving into ISAPI Programming* `<http://www.microsoft.com/mind/0197/isapi.htm>`
8. *Apache API notes* `<http://www.apache.org/docs/misc/API.html>`
9. *Oracle Web Application Server* `<http://www.oracle.com/st/o8collateral/html/xweb6ds.html>`
10. Boutell, T., 1996, *CGI Programming in C & perl*, Addison-Wesley, Reading (MA).
11. *Extended Tcl* `<http://www.neosoft.com/tclx/>`
12. *TclPro Debugger* `<http://www.scriptics.com>`
13. Libes, D., 1993, A Debugger for Tcl Applications, *Proceedings of the 1993 Tcl/Tk Conference, (PostScript)* `<http://www.mel.nist.gov/msidlibrary/doc/libes93c.ps>`

NeoWebScript: Enabling Webpages with Active Content Using Tcl

Karl Lehenbauer
NeoSoft, Inc.
karl@NeoSoft.com

Abstract

NeoWebScript marries the world's most popular webserver, Apache, with the Tcl programming language to create a secure, efficient, server-side scripting language that gives webpage developers simple-yet-powerful tools for creating and serving webpages with active content.

The ability to embed NeoWebScript code into existing webpages, without requiring a URL name change, leverages work done with webpage creation tools such as Netscape Communicator and Net Objects Fusion, while not disturbing links from remote sites and search engines.

A mature application, NeoWebScript-equipped webservers are currently in production on the Internet, serving real-world loads of millions of webpage "hits" per day.

This paper describes the driving forces behind the creation of NeoWebScript, and how those forces shaped its design and evolution into its current-day form. NeoWebScript's software architecture is described, and its capabilities are demonstrated using numerous examples, including webpage "visitor counters", rotating banner ads, queuing email, posting news, storing form submissions into Berkeley-style "dbopen" databases, and creating graphical images on the fly.

Finally, NeoWebScript's current status is summarized, our near-term plans are detailed, and conclusions are drawn.

1. Why Create NeoWebScript?

Our company, NeoSoft, Inc., is a large regional Internet Service Provider (ISP). NeoSoft has been running a webserver since 1992, and has been providing web-related services for customers for several years. The mix of customers using our web services spans the widest level of abilities, covering a broad range of applications. Customers use every platform and development tool, and every web publishing technique has limitations that affect how the developer interfaces with specific server-side capabilities. While much attention has been given to Java and JavaScript for client-side scripting, many forms of active content require that data be maintained, accessed and updated on the server. Such applications include electronic commerce, shared databases... even simple things like hit counters.

As our customers became more sophisticated and creative, they began asking for *active content* features such as access counters and rotating banner ads. Many had CGI scripts that they had written, or obtained, and wanted to run them on our webserver. Others wanted us to write them. We found that CGI programs had a number of problems:

- The overhead of the CGI approach is fairly high. For each CGI executed, the webserver must set up, start, manage, and communicate with a child process that executes the CGI.
- Redirecting existing URLs to scripts required reconfiguring the webserver. Redirected URL names tended to be unwieldy, and the rewritten web page addresses were confusing to users.
- Letting untrusted users run CGIs with the same user ID as the webserver creates a security problem; likewise there are security problems with running the webserver as superuser so that it can obtain permissions of the user when running a script as the user. A user could, in either case, accidentally or intentionally compromise their own files; in the latter case, an intruder may be able to directly gain superuser privileges.
- CGI programs typically emit the entire webpage programmatically, either rendering HTML authoring tools unusable or requiring "hand-stitching" to weave the tool-created HTML into the program, a fairly expert task that must be performed every time the tool-authored HTML is altered.

We began by modifying the webserver in C to add support for hit counters and banner ads. This approach was successful in that it provided some capabilities customers were asking for, and had lower overhead and was clearly more secure than the CGI approach, but it was immediately clear that we needed a more general solution than modifying the webserver on an ad-hoc basis. We wanted to create and make available to our users, and others, a simple and easy-to-learn tool for scripting active content.

To address these needs, we decided to integrate our own solution by mating an existing webserver with an embeddable scripting language.

We chose the *Apache webserver*[1] because we were already using it successfully in production to provide virtual web services for hundreds of domains. We had some skill with it, and we knew it to be capable, under active development, and in widespread use -- indeed, Apache is the world's most popular webserver[2] and comes with full source code. Since Apache is freely redistributable, including for commercial resale, it kept our options open with regard to creating a commercial version if we chose to. Derived from the widely used *NCSA http* server[3], among Apache's enhancements were a modular architecture, designed and documented for the purpose of accepting third-party "plug ins" such as the one we hoped to create.

For the scripting language we chose the Tool command language (Tcl)[4]. An explicit design goal of Tcl was that it be easy to integrate into other applications. Tcl is known for being easy to learn, and has strong text processing capabilities. We already had substantial experience with Tcl, which lowered the technical risk. Tcl's developer community had produced many excellent tools and packages, usually distributing them under the permissive Berkeley copyright, which we could utilize to add capabilities to the server that were far beyond our own resources to create and maintain. Finally, "Safe Tcl", by then supported as part of the Tcl core, looked very promising for providing users with a way to program the webserver with far less risk than the traditional server-side programming technologies.

2. Requirements

Fundamentally, NeoWebScript had to be reliable. It had to be able to serve millions of hits per day without failure. Accidental overwrites of adjacent data, memory leaks, etc, by the Tcl interpreter would have more serious consequences than for a CGI program, as the interpreter would run within the Apache server's child's address space, and a single Apache child process handles many webpage requests before terminating. Such problems wouldn't always manifest themselves until a subsequent page was served, which could significantly complicate debugging. Likewise, bugs in the Apache server could corrupt Tcl. Fortunately, both Apache and Tcl were (and are) robust and reliable – We have not had any significant problems with Tcl-enabled Apache processes malfunctioning or dumping core.

Another requirement was that, as our code progressed from an experiment to a production web scripting system, we embrace Apache's configuration files and configuration technology, which we did. All NeoWebScript-specific capabilities are configured and controlled through compliant configuration file extensions, using Apache's configurable module architecture.

One requirement greatly influenced the evolution of NeoWebScript. As an ISP, we have thousands of users who are not employees, all of whom are potential NeoWebScript developers, where *their* scripts are running on *our* servers. Compare this to a more traditional organization where, for example, a handful of employee-developers produce the organization's Internet and/or Intranet content. This is a critical distinction. As a result, NeoWebScript was designed from its inception to maintain the webserver's security while operating with an *untrusted user base*. This led to a design choice between providing webpage developers with the full power of a normal Tcl interpreter and our need to protect the server from those same developers, most of whom we have never met face-to-face.

Therefore, at every decision point, we opted for security over unencumbered power. By default, NeoWebScript's user files are kept out-of-band from the users' webpage files, protecting those files from being overwritten by a script, at the cost of not being able to script to create or manage files in the user's home directory. External programs cannot be executed except for certain specific applications (posting news and sending electronic mail, for instance) which are handled through tightly controlled interfaces that must be custom-developed for each supported program.

3. Design Philosophy

Our philosophy was that we would strive to make it easy to do "90%" of the things people wanted to do. This approach had served Mark Diekhans and I well in the development of Extended Tcl (TclX)[5], where, for example, we invented a simple way to create both client and server TCP sockets (which provided the model for what later became the Tcl core's *socket* command). This let users connect with or write Tcl code to talk to mail, news, the web, etc, but left the creation and interpretation of the less commonly used datagram (UDP) and more exotic multicast packets to other extensions such as Tcl-DP[6].

While our plan was to build in lots of capability, we wanted something that a fledgling webpage developer, with decent HTML skills but little or no prior programming experience, would be able to use to create simple active content features. A number of demos would be included, enabling web developers to cut and paste a number of interesting active-content elements into their webpages, all the while leaving their choice of HTML development tools completely open. An example of NeoWebScript usage appears in Figure 1.

```
<html>
<body bgcolor=white>
<h1>Welcome to my webpage</h1>
...
You are visitor number
<nws> incr_page_counter </nws>
...
</html>
```

Figure 1 - NeoWebScript HTML code fragment to produce a webpage "hit" counter that automatically increments every time the page is retrieved.

A major goal was to make it easy to receive, store, locate and recall data entered via forms. An example page that obtains and stores data sent in through a form is shown in Figure 2.

```
<title>Simple Form Result</title>
<h1>Simple Form Result</h1>
<nws>
load_response response
dbstore simple $response(name) response
</nws>
Response stored.  Thanks!
```

Figure 2 - The minimal NeoWebScript page to store the results of a form submission. By setting a form's action to point to a page containing this code, the key-value pairs comprising the form are stored in a Berkeley-style "dbopen" file[7], using the field "name" as the key.

Finally, for experienced developers, we planned to provide interfaces to database back-ends, a way to make and/or modify graphical images from data, provide access to the environment variables normally available to CGIs, and do whatever else users could do with the general-purpose programmability of a Safe Tcl interpreter. This would allow developers to customize content based on browser type and version, date, address, host name of the client, etc.

3.1 Commerce Servers

We also wanted NeoWebScript to have a secure socket layer (SSL) capability. This would allow NeoWebScript applications to support the encrypted sessions required of commerce applications, etc. The commercial *Stronghold* commerce server (http://www.c2.net) adds a secure socket layer (SSL) encryption capability to Apache, and would be used to create a compatible, secure version of NeoWebScript.

Another commerce server option is the freely redistributable *Apache-SSL*.[8] Although at one point no Netscape or Explorer-recognized certificating authorities would sign digital certificates for Apache-SSL, Thawte (http://www.thawte.com/) has been doing so for some time. With Verisign's (http://www.verisign.com/) recent decision to sign certificates for Apache-SSL, Apache-SSL represents another viable commerce server platform that is NeoWebScript-compatible.

4. How It Works

When a webpage containing NeoWebScript code is requested, an interpreter is created and loaded up with services and an array of information about the connection.

The embedded code is evaluated within the interpreter. A number of services are provided:

- A Safe Tcl interpreter
- Form elements included with pages sent as GET and POST requests can be imported into an array of key-value pairs. (An example is shown in Figure 2.)
- Arrays of key-value pairs can be written to and read from btree-indexed disk files with a single statement.
- Arrays of key-value pairs can be translated to SQL statements and written to Oracle[9] and Postgres[10] databases, on the local machine or across the network. SQL queries and cursor walks produce data as arrays of key-value pairs.
- Creation and/or modification of GIF-format graphic images from a Tcl script, using Thomas

Boutelle's *graphics draw* (gd) package.[11] Examples of on-the-fly graphic image generation can be seen in Figure 9 and Figure 11.

- Access to the environment variables normally available to CGI programs through the *webenv* array.

4.1 Apache Module Architecture

The Apache webserver is written in C. It has a modular architecture that splits the process of serving a webpage into eight stages. The stages are shown in **Table 1**.

URL-to-filename translation
Authentication
Authorization
Permissions
MIME Type Determination
Last-Stage Fixups
Emitting the Page
Logging the Results

Table 1 - Apache webpage pipeline modular stages

Apache provides a number of services to modules, controlled by a "switch" structure. Modules have the option of having initialization code executed at webserver startup time, of receiving configuration information when retrieving a page from config files in that page's directory, or merged among zero or more higher-level directories. Modules can opt to receive configuration information from the server config files, and can define handlers for one or more MIME data types.

Apache modules can install their code into a "chain" of handlers that can accept, pass on, or reject the aforementioned filename translations, user authentication and authorization checks, etc.

4.2 Processing Webpages

The NeoWebScript module is activated only when a request for a file matches the administrator-configured NeoWebScript file extension, which can be **.html** to enable any existing HTML page to include NeoWebScript code without a URL change, or a special extension such as **.nhtml**. If the file being requested does not match the special NeoWebScript MIME type, the file is processed by Apache's standard handlers, without any intervention by our code. If the page matches NeoWebScript's MIME type, it is handled in the same manner as Apache's server-side include handler (The NeoWebScript module is, in fact, based on Apache's *mod_include* module.)

Upon the first occurrence of a NeoWebScript tag, a safe interpreter is created and populated with variables, procedures and exported commands for use by the Tcl code contained in the webpage.

The embedded Tcl code is then evaluated, and its output is merged with any static content (standard HTML) that may be present in the page.

The same interpreter that executes the first chunk of NeoWebScript code executes any subsequent NeoWebScript code contained within that page.

After the page has been completely emitted, the interpreter is destroyed, as it cannot safely be reused. Note that the cost of creating, configuring and destroying a Tcl interpreter is far less than starting up and running a separate CGI program.

If no NeoWebScript tag is found in a page, the page is emitted without creating an interpreter or executing any Tcl code. This makes a NeoWebScript-enhanced Apache server perform at just a tick under the performance of an unenhanced server. When the NeoWebScript module is added to Apache, the server's memory footprint grows by the size of the NeoWebScript module, plus a Tcl interpreter and associated code. The processing overhead increases slightly due to the small overhead of looking for NeoWebScript tags.

As NeoWebScript's capabilities have evolved, our use of Apache's services has increased. Currently we support per-directory and merged-directory configuration, general configuration through the server config files, and, of course, handlers to parse and send pages by executing the NeoWebScript code embedded in the page and merging it with any static text present in the HTML file.

4.3 Supervisor Mode Pages

After we had NeoWebScript up and running for a while, we saw two things. One was that sites with a trusted user base – a relatively small number of developers who were trusted by the server administrators – were unnecessarily inconvenienced by not having all of the capabilities of a full Tcl interpreter. Another was that users were running into the need to develop their own *control pages*, i.e. pages that made it possible for them to see things like what db files they had, how big they were, etc.

We wanted to provide a way to allow trusted developers to have the full capability of the Tcl shell, even on a server where most developers were untrusted and hence would only have the safe functionality. Also, we wanted to be able to create special pages that could be accessed by many different users, where each user would use the same page to look at their files, but not at one another's.

Supervisor Mode gives pages in specified directories full Tcl capabilities (not just safe ones) and allows those pages to assume the identity of other users (with respect to the server-maintained user identities – to the operating system, all the NeoWebScript files are owned by the webserver). This allows construction of site-wide control pages, multi-user data upload and download pages, etc.

We added the ability for supervisor mode pages to do password authentications against the UNIX password file, allowing users to login to services provided by supervisor mode pages and using those pages to access and manage their NeoWebScript files.

4.4 Tcl-oriented logging module

We created a new logging module, *mod_log_neo*, to augment or replace the default logging module, *mod_log_config*. Our logging module combines access logs and browser-type logs. It avoids the relatively costly per-hit time and date conversions by logging times in UNIX integer-since-1970 format. It also writes logfile entries as Tcl lists, making them easier to parse by Tcl-based reporting tools. (This also makes it harder to spoof or trick the logfile processor with bogus URL requests.) An example of a message logged by *mod_log_neo* appears in Figure 3.

```
901018069 203.162.3.234 {} {} 304  0
vnmusicawards.com
{GET /summit2.jpg HTTP/1.0}
{Mozilla/4.05 [en] (Win95; I)}
```

Figure 3 – An example log entry produced by mod_log_neo (broken into multiple lines for readability). Entries are logged as Tcl lists consisting of the time, the IP address of the requesting host, the hostname of the requesting host (empty if IP-to-hostname DNS lookups are disabled), and the user ID (empty if none was specified). Next comes the HTTP status code and number of bytes returned, followed by the virtual host and HTTP request serviced, followed by the browser identification string.

The *NeoWebStats* application creates summary data from the logfiles, which average more than 100 megabytes per day for a site serving a million hits per day. Summaries can be combined to produce summaries spanning multiple days. NeoWebScript's on-the-fly graphics generation capability is used to produce pie charts showing the proportion of hits at various "depths" within the webserver. An example of NeoWebStats output is shown in Figure 11.

5. Examples

5.1 Retrieving Data from a Client Request

There are two common ways for web clients to send data to a webserver. One way is to use the HTTP[12] GET method, in which key-value pairs are coded into the URL request. The other is to use the POST method, in which the key-value pairs are sent following the HTTP request line and the key-value pairs (browser type, cookies, etc.) that are always sent, regardless of the type of request.

In either case, NeoWebScript parses the data in the request and stores it into a global Tcl array using the NeoWebScript command **load_response.** An example using **load_response** is shown in Figure 2.

5.2 Sending Email from a Webpage

An example of the use of **open_outbound_mail** to send an Internet email message from a webpage appears in Figure 4.

```
set fp [open_outbound_mail \
        "Moving to Montana?" $toWhom]
puts $fp "Do you think I could interest you in"
puts $fp "a pair of zircon-encrusted tweezers?"
close $fp
```

Figure 4 - Sending email from within NeoWebScript

Open_outbound_mail returns a Tcl filehandle. The embedded code uses **puts** and any other relevant file-oriented Tcl commands to create the message body. The email sender's address is automatically constructed from the username of the owner of the webpage that's being interpreted and the name of the server that did the serving. If *to* is not specified, the recipient is also set to be the user name of the owner of the webpage.

5.3 Posting News

Open_post_news starts a Usenet news posting, returning a Tcl filehandle to be used with **puts**, etc, to specify the contents of the body of the news posting. The message always comes "from" the user name of the owner of the webpage being interpreted, coupled with the name of the server doing the serving. Example code for posting news is shown in Figure 5.

After writing out the body of the news article, a line is written to the filehandle consisting of a single period, then the file is closed. (This is a requirement of NNTP. It could, of course, be hidden by a proc.)

```
set fp [open_post_news -subject $subject \
    -newsgroups neosoft.announce \
    -distribution neosoft]
puts $fp "This is the body of the message."
puts $fp "."
close $fp
```

Figure 5 – Posting news from NeoWebScript

Note that for this to work you must have a news server supporting Network News Transfer Protocol (NNTP)[13] in the same domain as your webserver. For example, within *neosoft.com*, **open_post_news** will contact the news server *news.neosoft.com.*

5.4 Graphical Access Counters

A "hit counter" for the current page is fetched, incremented, and returned by calling **incr_page_counter**. **Incr_page_counter** automatically manages the counter using a db-file, where the index is created based on the current URL. An example showing the number of times a page has been visited, where GIF files represent each digit of the count is shown in Figure 6.

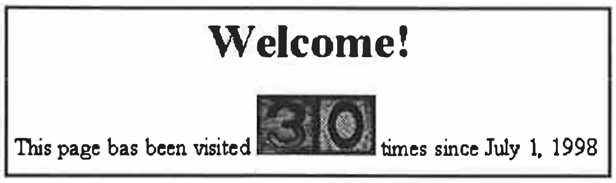


Figure 6 - Number of page visits shown by on-the-fly constructing HTML image references to images that exist as numbered GIF files, one per digit.

The code that created this display is shown in Figure 7. **Split** is used with an empty string for the characters to split on, causing the number of visits to be exploded into a list, where each digit in that number is one element of the list. We walk the list using **foreach**, emitting image references to a GIF image file for each digit to be displayed.

```
<center>
<h1>Welcome!</h1>
This page bas been visited
<nws>
foreach num [split [incr_page_counter] {}] {
html "<img src=/images/counters/set22/$num.gif>"
}
</nws>
 times since July 1, 1998
</center>
```

Figure 7 - NeoWebScript code fragment for the graphical access counter example.

5.5 Rotating banner ads

Using Tcl, rotating banner ads can be easily implemented by calling **random_pick_html**. Figure 8 is a real example from the NeoSoft, Inc. homepage, located at http://www.neosoft.com/. The example has been simplified by having the heights and widths of the images, and alternate image strings removed to improve readability.

```
random_pick_html {
{<a href=/neosoft/>
        <img src=images/sroom2.gif></a>}
{<a href=/neosoft/neorules.html>
        <img src=images/resources.gif></a>}
{<a href=/neopolis/>
        <img src=images/neopolis.gif></a>}
{<a href=/neowebscript/>
        <img src=images/logos/nws7.gif></a>}
}
```

Figure 8 - NeoWebScript code demonstrates the random selection of one of a number of specified HTML components every time the page is retrieved.

5.6 Issuing and retrieving cookies

```
neo_make_cookie -user ellyn -days 30 \
        -path /myApp
```

This example creates a cookie named *user* containing the text *"ellyn"*. This cookie will be sent by the user's browser as part of all HTTP requests sent to this same server, for a period of 30 days, whenever the URLs requested are underneath `/myApp` on this server, and the browser is cookie-enabled.

```
load_cookies cookies
html $cookies(email)
```

`load_cookies` retrieves all of the cookies, if any, uploaded by the browser in the HTTP request header, breaking out each cookie into a separate array element of the array name specified in the call.

5.7 Creating Graphic Images from NeoWebScript

NeoWebScript can generate and/or modify graphic images at runtime. In Figure 9 we see an "analog" clock, generated live, showing the current time of day.

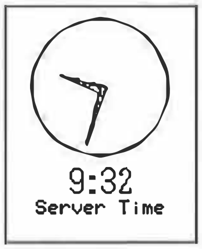


Figure 9 - GIF file of an analog clock, produced on the fly using NeoWebScript

We create live GIF files by creating Tcl scripts that are end in a **.gd** extension. When the file is referenced from an **<img src=...>** tag in a webpage, our code is executed and is expected to produce a GIF file when complete by executing *gd writeGIF*. (Note that the image filehandle is passed in as the hard-coded global variable *imageFile*.)

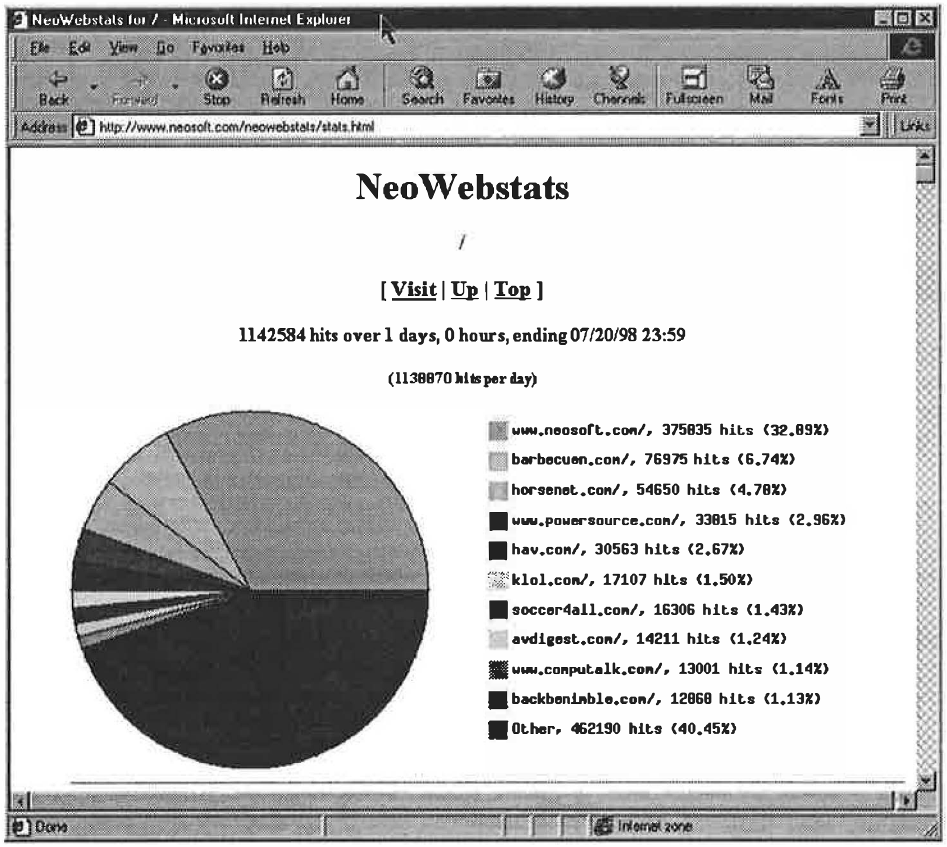
```
proc draw_analog_time {im color} {
    scan [clock format [clock seconds] -format "%I %M"] "%d %d" hour minute
    set hourStart [expr 90 - ($hour * 30 + $minute / 2)]
    set minuteStart [expr 90 - ($minute * 6)]

    set minuteX [expr int(40 + cos($minuteStart * 3.14159 / 180) * 30)]
    set minuteY [expr int(40 - sin($minuteStart * 3.14159 / 180) * 30)]
    gd line $im $color 40 40 $minuteX $minuteY; # draw three times
    gd line $im $color 40 43 $minuteX $minuteY; # to make it pointy
    gd line $im $color 43 40 $minuteX $minuteY

    set hourX [expr int(40 + cos($hourStart * 3.14159 / 180) * 20)]
    set hourY [expr int(40 - sin($hourStart * 3.14159 / 180) * 20)]
    gd line $im $color 40 40 $hourX $hourY
    gd line $im $color 40 43 $hourX $hourY
    gd line $im $color 43 40 $hourX $hourY

    gd text $im $color large 25 80 "[format %d:%02d $hour $minute]"
}

set im_out [gd create 82 110]; # create the image
set white [gd color new $im_out 255 255 255]; # background
set black [gd color new $im_out 0 0 0]

gd arc $im_out $black 40 40 70 70 0 360; # draw clock face
gd text $im_out $black small 8 95 "Server Time"
draw_analog_time $im_out $black
gd writeGIF $im_out $imageFile
```

Figure 10 - NeoWebScript code for making the analog clock GIF file

The code that produced the analog clock is shown in Figure 10. In this code, the dimensions of the GIF file are specified using *gd create.* Colors are allocated, then we draw a circle to represent the clock face. We use *gd text* to write the text "Server Time" to the image, then call the proc **draw_analog_time** to create the clock hands and write the numeric time into the image.

Draw_analog_time fetches the current time in hours and minutes into variables called *hour* and *minute.* A bit of trigonometry serves to calculate the endpoints of the clock hands. By drawing the hands three times, each with slightly different locations for the "center" of the clock face, the clock come out "thicker" in the center of the clock face, drawing to a point at the ends.

A more sophisticated instance of graphics generation from NeoWebScript can be found in *NeoWebStats* (see Figure 11), an on-demand graphic image generator that creates pie charts to show what areas of a website are getting what proportion of hits.


NeoWebstats for / - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History Channels Fullscreen Mail Fonts Print
Address http://www.neosoft.com/neowebstats/stats.html
Links
Done
Internet zone

NeoWebstats

/

[Visit | Up | Top]

1142584 hits over 1 days, 0 hours, ending 07/20/98 23:59

(1138870 hits per day)

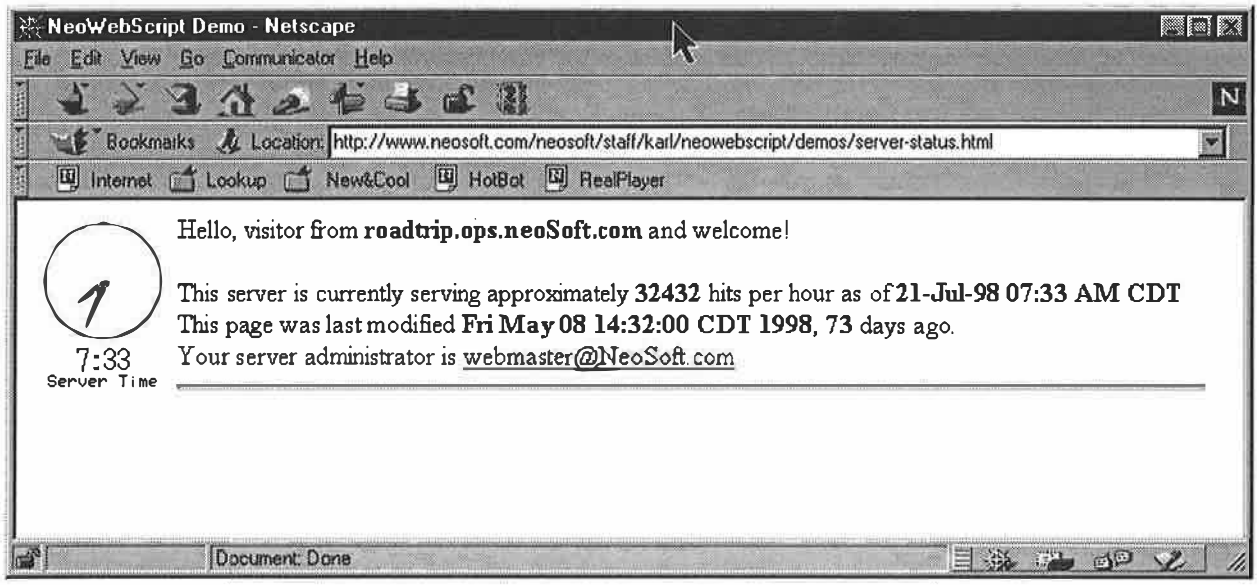
- www.neosoft.com/, 375835 hits (32.89%)
- barbecuen.com/, 76975 hits (6.74%)
- horsenet.com/, 54650 hits (4.78%)
- www.powersource.com/, 33815 hits (2.96%)
- hav.com/, 30563 hits (2.67%)
- klol.com/, 17107 hits (1.50%)
- soccer4all.com/, 16306 hits (1.43%)
- avdigest.com/, 14211 hits (1.24%)
- www.computalk.com/, 13001 hits (1.14%)
- backbeninble.com/, 12868 hits (1.13%)
- Other, 462190 hits (40.45%)

Figure 11 - On-the-fly graphical image generation is used to show the relative proportions of hits among a number of virtual sites. Visitors have the ability to "drill down" into a site and see the proportions of hits among subdirectories within each site, up to a settable maximum depth.

5.8 Accessing Web Environment Variables

NeoWebScript makes approximately thirty Apache webserver environment variables available to the Tcl interpreter when it's interpreting a NeoWebScript page. Among these are the referrer URL (HTTP_REFERER), the browser type and version (HTTP_USER_AGENT), plus a small number new ones, including the last modify date of the document (NEO_LAST_MODIFIED), and the user ID of the owner of the document being served (NEO_DOCUMENT_UID). These environment variables can be accessed by NeoWebScript code through the global *webenv* array. An example webpage that displays some data contained in, and derived from, the web environment array is shown in Figure 12.

The HTML code that created the display is shown in Figure 13. **Remote_hostname** returns the name of the host requesting the page, if it can be determined. (Otherwise the IP address is returned.) **Estimate_hits_per_hour** returns an estimate of the number of hits being served per hour. It does this by examining the server's **access_log** file. It works by seeking approximately 1,000 hits backwards into the access log, comparing the time that entry was made to the current time, and extrapolating the estimated number of hits per hour that the webserver is serving. We then format the current time and the date when the webpage was last modified using the standard Tcl **clock** command. Finally we emit an HTML[14] **mailto:** link to the web administrator, as read from the `SERVER_ADMIN` variable.


NeoWebScript Demo - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://www.neosoft.com/neosoft/staff/karl/neowebscript/demos/server-status.html
Internet Lookup New&Cool HotBot RealPlayer
7:33
Server Time
Document: Done

Hello, visitor from **roadtrip.ops.neoSoft.com** and welcome!

This server is currently serving approximately **32432** hits per hour as of **21-Jul-98 07:33 AM CDT**
This page was last modified **Fri May 08 14:32:00 CDT 1998**, **73** days ago.
Your server administrator is webmaster@NeoSoft.com

Figure 12 – Webpage showing data derived from webserver environment variables.

```
<img src=gdclock.gd align=left>
Hello, visitor from <b>
<nws>html [remote_hostname]</nws></b>
and welcome!

<p>This server is currently serving
approximately <b>
<nws>html [estimate_hits_per_hour]</nws>
</b>hits per hour as of <b>
<nws>html [clock format [clock seconds] \
        -format "%d-%b-%y %I:%M %p %Z"]</nws>
</b><br>This page was last modified <b>
<nws>html \
"[clock format $webenv(NEO_LAST_MODIFIED)]"
</nws></b>, <b>
<nws>html "[expr ([clock seconds] - \
  $webenv(NEO_LAST_MODIFIED)) / 86400]"</nws>
</b> days ago.<br>Your server administrator is
<nws>
html "<a href=mailto:$webenv(SERVER_ADMIN)> \
    $webenv(SERVER_ADMIN)</a>"
</nws>
```

Figure 13 – HTML with embedded NeoWebScript that created the webpage shown in Figure 12

5.9 Subst-style NeoWebScript Pages

An alternative to invoking NeoWebScript code within **<nws>** and **</nws>** tags is server-subst-style NeoWebScript. With subst-style code, the entire HTML file being served is run through Tcl's **subst** command, causing dollar sign variable substitution and square-bracketed code to be evaluated, with the results substituted in place.

The webserver's ability to evaluate subst-style webpages is configured as a handler via Apache's **srm.conf** file. The **AddHandler** directive allows you to map certain file extensions to "handlers", which causes Apache to take actions based on file extension. For example, to cause files ending in **.shtml** to be Tcl-substituted and emitted as a **text/html** MIME type, the following lines must be enabled in the **srm.conf** file:

```
AddType text/html .shtml
AddHandler server-subst .shtml
```

A subst-style implementation of the code that produced the webpage in Figure 12 is shown in Figure 14.

```
<img src=gdclock.gd align=left>
Hello, visitor from <b>[remote_hostname]</b> and welcome!

<p>This server is currently serving approximately <b>[estimate_hits_per_hour]</b> hits per hour as of <b>
[clock format [clock seconds] -format "%d-%b-%y %I:%M %p %Z"]

</b><br>This page was last modified <b>[clock format $webenv(NEO_LAST_MODIFIED)]</b>, <b>
[expr ([clock seconds] - $webenv(NEO_LAST_MODIFIED)) / 86400]</b> days ago.
<br>
Your server administrator is  <a href=mailto:$webenv(SERVER_ADMIN)>$webenv(SERVER_ADMIN)</a>
```

Figure 14 - A NeoWebScript subst-style webpage that produces the same results as the HTML shown in Figure 13. Note the lack of <nws> tags – the entire page is interpreted using Tcl's **subst** command, causing in-place variable substitution and evaluation and substitution of square-bracketed code.

6. Current Status

There are currently about 1,200 sites running NeoWebScript, according to the Netcraft survey. [15]

6.1 Availability

The current version of NeoWebScript is available for download from http://www.neosoft.com/neowebscript.

6.2 Version 3.0

A new version of NeoWebScript, version 3.0, is currently in beta. This release is the first one to integrate Tcl 8.0 (Version 2.3 uses Tcl 7.6), yielding significant performance improvements in most cases. Also it builds against Apache 1.3.1, and uses *dbopen 2.4.14.* Version 2 of dbopen has new locking, logging and transaction-oriented capabilities, and the release includes new code to support those capabilities through their native version 2 interfaces.

The 1.3.1 version of Apache is much easier to build, using a standard GNU (ftp://prep.ai.mit.edu/pub/gnu) *configure* script to automatically configure the package for the capabilities of the particular UNIX system for which it's being compiled. Apache 1.3.1 also includes a compiler "driver" that greatly simplifies building third-party modules. Also, it allows those modules to be kept separately from the Apache core, and NeoWebScript is currently being built and tested using this technology.

Apache 1.3 includes support for Windows 95 and NT for the first time, and work is underway to make NeoWebScript run in those environments as well.

6.3 Making It Easier to Build

Although we have steadily decreased the amount of effort required to build NeoWebScript from source code, we continue to add capabilities and interface with additional packages. A successful build of NeoWebScript requires successful builds of Apache, Tcl, TclX, and NeoTcl (A NeoSoft-maintained Tcl extension set.) Now that we have added interfaces for Postgres, Oratcl, Scotty, and other packages, we felt that, overall, we were losing ground in this area – NeoWebScript can be fiendishly difficult to build, configure, and install. A binary release for Windows, and native install packages for popular Unix architectures, will make NeoWebScript much easier to get running, continuing our work to improve the build process. These improvements will bring NeoWebScript to an entirely new audience. As always, we will build on the excellent work done by others in the Tcl community wherever possible.

6.4 Cgi.tcl

Although I have stated a number of concerns about the CGI method of producing webpages from programs, I'd like to point out that Don Libes' *cgi.tcl*[16] package provides powerful tools for generating sophisticated HTML constructs from within Tcl. NeoWebScript and cgi.tcl not only play together – developers using cgi.tcl's capabilities from within NeoWebScript have powerful tools for simplifying and enhancing their Tcl-generated HTML content.

6.5 Year 2000 Issues

We do not anticipate any "Year 2000" problems with NeoWebScript itself, as both Tcl 7.6 and Tcl 8.0's date functions are y2k-safe. The Apache group claims to have worked through all of their year 2000 issues. We have budgeted time for testing year 2000 issues in our current release cycle.

6.6 Future Development

Future development interests include creating tools to simplify providing a uniform look across a set of pages. Website integration tools are also an area of interest – NeoWebScript is a natural platform for integrating site-oriented search engines, validating links, etc.

7. Conclusions

NeoWebScript was designed to support NeoSoft, Inc. in providing server-side scripting capabilities to an untrusted user base while providing us with a margin of safety while doing so. As such, NeoWebScript is of particular use to ISPs, web-hosting providers, Free-Nets, etc. After two and a half years of use, there have been no known security breaches related to exploits involving NeoWebScript.

Web content developers, including those without significant prior programming experience, have enthusiastically received NeoWebScript.

Sizable applications have been written in NeoWebScript, including two shopping carts, an editable event calendar and to-do list, in-out board, a web-based chat system, and a Yahoo-like hierarchical organizer (http://www.ghofn.org), among many others, all using our standard safe-interpreter interfaces. In fact, NeoWebScript has all but eliminated the need for other server-side tools, by allowing us to quickly and easily develop NeoWebScript equivalents for virtually every other commercial web-related application we have seen.

"Supervisor mode" provides a way to bypass the limits enforced by the architecture of the safe-interpreter/Apache interface in environments populated with a trusted user base, providing trusted developers with significantly more power than they would otherwise have had. Supervisor mode offers many as-yet-unexplored possibilities, and does not yet provide as tight of an integration of capabilities as is seen on the safe side.

A sizable NeoWebScript code base now exists, and we must balance our desire to innovate with the need to maintain compatibility with the work that has already been done.

NeoSoft is committed to continuing to develop and maintain NeoWebScript. We have been able to leverage this work in our own internal intranet applications, use it to lure, win, and keep web developers and the customers they bring with them, and to successfully win and deliver on a high-profile extranet site for a Fortune 100 company.

Hooking Tcl up with Apache is a simple idea that was straightforward to implement. So far, it has been one of the differentiators that have given us an edge in a highly competitive business.

[1]Apache Server Project, http://www.apache.org/

[2] Netcraft Webserver Survey, http://www.netcraft.co.uk/survey/

[3] NCSA httpd webserver, ftp://ftp.ncsa.uiuc.edu/Web/httpd/Unix/ncsa_httpd/

[4] John Ousterhout, http://www.scriptics.com/people/john.ousterhout/

[5] TclX 8.0, Mark Diekhans and Karl Lehenbauer, http://www.neosoft.com/tclx

[6] Tcl-DP: a distributed programming extension to Tcl, http://simon.cs.cornell.edu/Info/Projects/multimedia/Tcl-DP/

[7] The Berkeley dbopen Database, Keith Bostic, http://www.sleepycat.com/db.download.html

[8] Apache-SSL secure webserver, http://www.apache-ssl.org/

[9] Tom Poindexter, Oracle/Tcl Interface, *oratcl*, http://www.nyx.net/~tpoindex/tcl.html

[10] POSTGRES SQL database, http://www.postgresql.org/

[11] Thomas Boutell's gd graphics library, http://www.boutelle.com/gd/

[12] HTTP Protocol Specification, RFC1945 - Hypertext Transfer Protocol – HTTP/1.0, http://www.cis.ohio-state.edu/htbin/rfc/rfc1945.html

HTTP Protocol Specification, RFC2068 - Hypertext Transfer Protocol – HTTP/1.1, http://www.cis.ohio-state.edu/htbin/rfc/rfc2068.html

[13] NNTP Network News Transfer Protocol Specification, RFC-977, http://www.cis.ohio-state.edu/htbin/rfc/rfc977.html

[14] HTML Specification, http://www.w3.org/pub/WWW/MarkUp/Wilbur/

[15] Netcraft Webserver Survey (Ibid.)

[16] cgi.tcl, Don Libes, http://expect.nist.gov/cgi.tcl/

XML Support For Tcl

Steve Ball
Zveno Pty Ltd
http://www.zveno.com/
Steve.Ball@zveno.com

Abstract

XML is emerging as a significant technology for use on both the World Wide Web and in many other application areas, such as network protocols. Documents written in XML have a rich, hierarchical structure, the document tree. An application which is to process XML documents must be able to access and manipulate the document tree in order to be able to examine and change the structure.

The DOM is a language-independent specification of how an application accesses and manipulates the document structure. TclDOM is a Tcl language binding for the DOM. The TclDOM specification provides a standard API for Tcl applications to process a XML or HTML document.

TclXML is a Tcl package which provides a sample implementation of TclDOM. It provides XML parsers along with the tools needed to create a hierarchical representation of documents which can be conveniently processed by a Tcl script. There are also facilities to check the validity of a document, along with commands to produce document output. TclXML provides a framework for parser and validator modules which allows some or all of the various components to be implemented in an extension language.

Keywords: Tcl, World Wide Web, WWW, XML, DOM, Parsing

A Brief Introduction To XML and DOM

XML

The eXtensible Markup Language, XML [XML], is a Recommendation from the World Wide Web Consortium (W3C) [W3C] which is a significant simplification of SGML, intended to make a subset SGML suitable for use on the Web. XML provides a much more meaningful and rich way in which to represent semantic structure in a document when compared to HTML. With XML, a document author can "call a Spade a <Spade>". That is to say, elements can be given names which are meaningful to the author and convey more of the semantics of the document. These elements are defined using a Document Type Definition (DTD), along with allowed general entities. The DTD includes rules which govern what elements or text are allowed to occur inside each element. In contrast, HTML gives the author a fixed set of tags to use, and those tags have fixed semantics and behaviour.

XML has been designed with the goal of being easier for tool developers to write software for processing documents, so that there will be plentiful applications available for handling XML. The syntax is far more restricted than SGML, with most optional features removed. For example, tag minimisation and omission are not allowed so it is easy to detect the end of an element. Empty elements must be explicitly marked.

There are two levels of conformity for a XML document. At the very least, a XML document must be *well-formed.* A well-formed XML document adheres to all of the syntactic rules of the XML specification. All elements must have an end tag, unless they are empty elements in which case the tag must include a trailing slash, such as <BR/>. Element attributes must have a value and the value must be quoted. There may be no occurrances of illegal characters, and so on. XML has been designed so that a program can check that a document is well-formed without having to use the document's DTD.

In addition to being well-formed, a XML document may also be *valid.* A valid XML document conforms to the rules laid out in its associated DTD. This means that all elements have content which is allowed according to the element's definition of its content model in the document's DTD. All attributes used in elements must also be permitted for that element and their value must be of the correct type. All attribute values which are used for identification must be unique, and so on. Validity is a much stronger assertion for a document than well-formedness, but requires access to the document's DTD and more processing time to check the various rules.

XML is internationalised and support for Unicode is a requirement of XML, so Tcl version 8.1 is well positioned to be used for software applications which handle and process XML documents.

The following document is a small example of a XML document instance:

```
<?xml version="1.0">
<!DOCTYPE paper SYSTEM "paper.dtd">
<paper>
<title>TclXML Example</title>
<abstract>This is an example of a
XML document instance.
</abstract>
<introduction>XML uses the
<acronym><short>DTD</short>
<full>Document Type
Definition</full></acronym>
to define classes of documents.
Such a document is known as a
<definition>document
instance</definition>.
</introduction>
<point>A DTD is used to define the
elements and entities that are
allowed to appear in a document.
Empty elements have a trailing
slash:
<figure image="figure-1.png"/>
</point>
<conclusion>XML is way better than
HTML.</conclusion>
<references></references>
</paper>
```

DOM

The Document Object Model (DOM) [DOM] is a language-neutral specification of a standard Application Programming Interface (API) for both accessing the features and the content of a document and creating or modifying documents. Documents are in the form of a tree and the DOM has methods of accessing properties of the tree nodes. DOM provides a core set of features and specific features may be provided for certain markup languages. In the first instance, DOM is providing a specification of features for XML and HTML. The aim of the DOM is to allow application developers to write software which manipulates a document and to use whichever programming language is suitable for the task. The same underlying constructs will be available in whatever language the developer chooses.

DOM already has language bindings for Java and ECMAscript. Many tools are being written in Java for processing XML, and standard interfaces are starting to emerge for Java components, such as XAPI-J and SAXDOM. A language binding for Tcl is discussed in this paper.

Other Proposed Standards

XML will never replace HTML because XML does not supply any of HTML's semantic features, in particular the elements which provide presentational and/or behavioural semantics such as STRONG, EM, UL and FORM. HTML elements perform a function when they are used in a document, whereas XML elements do not. XML elements only provide syntactic declaration of a document's structure. Of course, this is what makes XML useful for a wider range of applications than HTML.

In order to provide certain semantics for the elements used in a XML document other facilities are required. A number of languages have been proposed for these purposes. The XML Link Language, XLL, provides the hyperlinking model for XML. XLL has been derived from TEI and HyTime and as a result will be a much richer model than HTML's. For example, XLL will allow bidirectional and multi-destination links, as well as allowing link targets to specify and use the structure of the destination resource. In order to render XML documents for viewing or printing a stylesheet language will be used. Either Cascading Style Sheets (CSS) [CSS] will be used or the new proposed XML Stylesheet Language (XSL). XSL will also include scripting.

Tcl Support For XML

Tcl can be very useful for processing and generating XML documents, just as Tcl is useful for handling HTML documents in CGI scripts. There are many applications which may be able to use a scripting language for document processing, both in GUI and non-GUI modes and in standalone, server and client environments.

Since applications will have varying requirements, different supporting libraries may be used to process documents. Some of these may be implemented entirely in Tcl, for cross-platform portability, some may use C or C++ extensions to improve performance and some may interface with Java components. Ideally, there will be a standard programming interface for applications written in Tcl to access the document structure for processing. It is the goal of TclDOM to provide that standard interface, based upon the DOM specification.

XML support in Tcl has two distinct parts. Firstly, TclDOM is a language binding for the DOM.

TclDOM provides an implementation-independent specification of a Tcl API of the DOM for Tcl scripts to access and manipulate XML and HTML documents. Secondly, TclXML is a sample implementation of TclDOM which provides XML parsers and generators.

Zveno is making the TclDOM specification and TclXML freely available. Both the TclDOM specification and the TclXML distribution may be found on the Zveno website [TCLXML]: `http://www.zveno.com/zm.cgi/in-tclxml/`

TclDOM

TclDOM is a Tcl language binding for the DOM. The specification details how to access DOM features using Tcl constructs. The goals for TclDOM are: to provide access to all features of the DOM from a Tcl script, to allow features from other DOM-compliant languages (such as Java) to be used in conjunction with Tcl scripting and finally to provide an interface which uses conventions familiar to Tcl developers.

Access to all of the features of the DOM has the advantage of familiarity to developers who have a previous knowledge of the DOM. The DOM is also a collaborative design effort which ensures that the interface will be comprehensive, allowing sufficient expressiveness of the interface for a general-purpose document scripting library such as TclXML.

Providing all of the DOM features goes a long way towards allowing access to DOM implementations in other languages, because there is then a straightforward mapping between the implementations. It is fair to say that the majority of work in writing XML processors is being done using Java, so an interface to Java is essential for TclXML to leverage this activity. Fortunately, TclBlend and Jacl give easy access to Java classes from Tcl scripts.

TclDOM Interface Design

Tcl and Tk use a number of conventions to make Tcl scripting easier for developers. TclDOM will adopt these conventions in the design of its API so that Tcl developers will find it easy to use. For example, using Tcl commands to manipulate objects and the use of configuration options. The design of the TclDOM API must take into consideration that an application may need to process more than one document concurrently, so a single hierarchical model, such as Tk's widget hierarchy, would be unsuitable. For example, a XML document browser may need to access the document's tree structure while at the same time accessing the document's XSL stylesheet, which is itself a XML document.

Previous Work

Uhler's `html_library`

Stephen Uhler wrote an all-Tcl HTML parser called `html_library` [Uhler95]. This library was sufficiently generalised internally to be easily adapted for parsing XML documents. Indeed, the tokeniser which is part of the Tcl XML parser in TclXML is derived from `html_library`.

`html_library` translates a HTML or XML document into a series of calls to a Tcl procedure. Hence, it is an event-based parser where the application is notified of the start and end of elements. The parser does no checking for well-formedness or validity - all further processing is left to the application. This parser does not provide a good model for TclDOM, since the DOM uses a tree-based view of the document structure.

Plume

Plume [Ball98A] is a general-purpose WWW browser. Plume version 1.0 (never officially released publically) extended the `html_library` parser to build a tree representation of the parsed HTML or XML document. This representation was then presented to the application with the tree structure, in a format known as "XAPI-Tcl", the XML API for Tcl. XAPI-Tcl used a nested Tcl list structure to represent the document tree, with certain commands used to distinguish between elements, character data, processing instructions, and so on. For example, the document example given above would be represented as:

```
parse:pi xml {version 1.0} {}
parse:pi DOCTYPE {SYSTEM paper.dtd}
{}
parse:element paper {} {
 parse:element title {} {
  parse:text {TclXML Example} {} {}
 }
 parse:element abstract {} {
  parse:text {This is an example of
a XML document instance.}
 }
 parse:element introduction {} {
  parse:text {XML uses the }
  parse:element acronym {} {
   parse:element short {} {
    parse:text DTD {} {}
   }
   parse:element full {} {
    parse:text {Document Type
Definition}
   }
  }
  parse:text {to define classes of
documents. Such a document is known
as a } {} {}
  parse:element definition {} {
   parse:text \
    {document instance} {} {}
  }
  parse:text . {} {}
 }
 parse:element point {} {
  parse:text {A DTD is used to
define the elements and entities
that are allowed to appear in a
document. Empty elements have a
trailing slash: }
  parse:element figure {image
figure-1.png} {}
 }
 parse:element conclusion {} {
  parse:text {XML is way better
than HTML.}
 }
 parse:element references {} {}
}
```

This format may be interpreted as a (nested) Tcl list or evaluated as a Tcl script. Internally the parser, `xml::parse`, manipulates the document as a Tcl list. Since in Tcl version 8.0 (and above) list traversal is relatively fast the parser also returns the parsed data structure as a Tcl list. However, this format is not suitable for evaluation so a command, `xml::cvtscript`, is provided to convert this representation into one which may be evaluated, as shown above.

To facilitate access to the data structure using Tcl list commands, in particular the `foreach` command, dummy arguments are appended to the entries used for text and processing instructions. This allows a construct to be used such as:

```
foreach {type arg1 arg2 arg3}
[xml::parse $MyDocument] {
  switch $type {
    parse:element {
      #   Process element
      # arg1 is the tag name
      # arg2 is the attribute list
      # arg3 is the element content
    }
    parse:text {
      #   Process character data
      # arg1 is the data
      # arg2 and arg3 are unused
    }
    parse:pi {
      #   A processing instruction
      # arg1 is the PI name
      # arg2 is data for the PI
      # arg3 is unused
    }
  }
}
```

As discussed above, an alternative way to process a document is to evaluate the parsed data structure. All arguments are appropriately quoted, so this is a safe operation to perform. In this case, the application defines procedures with the same name as those names used to distinguish the features of the document. By default, these are:

`parse:element`

: To denote an element, option `-elementcommand`.

`parse:text`

: To denote character data, option `-textcommand`.

`parse:pi`

: To denote a processing instruction, option `-picommand`.

`parse:comment`

: To denote a comment, option `-commentcommand`.

Different names may be used by specifying options to the `xml::parse` command. This processing method may be used as such:

```
proc parse:element {name attributes
content} {
    eval $content
}
proc parse:text {text unused
unused} {
    puts $text
}

eval [xml::cvtscript [xml::parse
$MyDocument]]
```

The disadvantage of using an explicit Tcl representation for the parsed data structure is that the opportunity is lost to implement the document tree using another language, such as C or Java. Accessing and manipulating the tree structure may be quite slow using Tcl. Also, it can be difficult to dynamically modify the document structure, for example for document editing purposes, and to navigate the tree from an arbitrary starting point, for example if a tree node is passed as an argument.

CoST

CoST [English96] is a system for processing SGML documents written by Joe English based on James Clark's `sgmls` library. It has been adapted for use with Tk in an application called CoSTWish [PM-R96], and is being used for processing XML documents for the DOM specification [Nicol98].

CoST's API is not directly suitable for use with TclDOM. CoST has the notion of a document node, but nodes are not exposed to the application and there is no equivalent to a DOM "Iterator" for traversing sequences or trees of nodes. Instead, CoST supplies explicit methods for performing iterative operations on document tree nodes, such as `withNode` or `foreachNode`. In addition, CoST does not provide support for creating or modifying documents, nor does it provide access to the document's DTD.

Another difference between DOM and CoST is the notion of a current node. In CoST there is always a node which is current, and operations may be performed upon that node by CoST methods. However, in the DOM no actual node is specified as being current, instead there is a pointer to the position between two nodes. The DOM then allows the node before or after the current pointer to be retrieved. The DOM specifies node positions in this way to allow traversal of a dynamically changing document tree. With the DOM there is no danger of the current node being deleted and so the current node reference becoming invalid.

The TclDOM API

A preliminary API for TclDOM has been designed to satisfy the constraints outlined above. The namespace `dom` shall be reserved for use by the TclDOM package. Layered packages may also be used for supporting specific markup languages. The namespaces `xml` and `html` are initially reserved for XML and HTML respectively.

DOM is defined using the OMG IDL interface specification, a language-neutral definition language. This presents some difficulties for defining the TclDOM specification because IDL is object-oriented, whereas Tcl is not. Common Tcl conventions are used to overcome this problem, by defining class creation commands and object instance commands. TclDOM defines a number of commands within the `dom` namespace which correspond to the IDL interfaces defined in the DOM specification. These commands produce and accept "tokens" as arguments for referring to nodes in the document tree. The DOM implementation may use these tokens to lookup the node in an internal data structure. This allows for efficient implementation in extension languages.

The DOM specification provides a class for accessing a list of nodes and a class to manipulate strings. These are unnecessary in Tcl, as Tcl already has a rich set of primitives for manipulating lists and strings, but these interfaces may be emulated for the sake of compatibility. TclDOM defines methods for retrieving a list of the children of a node, the parent of a node, the list of attributes for an element, and so on. An application may use these methods to traverse the document tree, much like a Tk script traverses the widget hierarchy. Tcl lists are ordered, which is an important property for representing the children of an element node. One potential problem with this approach is that in an application where the document is being dynamically updated the document tree may change after a list of nodes has been generated. However, this may also be seen as an advantage when compared to Plume's nested list approach where changing the document tree can be cumbersome.

Attribute lists are also defined in terms of a Tcl list, but are represented as name/value pairs. This is convenient for use in conjunction with the `array set` Tcl command for accessing attributes via a Tcl array. Attribute lists are unordered, so storing these in a Tcl array is satisfactory.

The command necessary to perform parsing and serialisation a XML document instance is not explicitly provided by the DOM specification. For TclDOM these functions may be made implicit by defining that XML documents are stored as an internal representation of a Tcl Object. In this way, a XML document will be stored initially as a string, but when accessed by a DOM function it will be parsed into the implementation's internal data structure. When the string representation is required the internal structure is serialised. The only problem with this approach is that an implementation would be difficult to write as a pure Tcl script, since the internal representation of a Tcl Object cannot be accessed from the Tcl script level.

Example

The following is an example of creating a document using TclDOM. This example creates the first few elements of the example given earlier, modifies the document and then saves the XML text in a file. Note that this interface is a preliminary one based on the 20th July 1998 draft version of the DOM Core specification.

```
# Implicitly parse a document
set text [dom::text cget -nodeValue
    [lindex [dom::node children
{<Example>Sample Text</Example>}]
0] ]

# Create a document in memory

set docRoot [dom::document
 createDocumentFragment]
set paper [dom::document
 createElement $docRoot paper]
set title [dom::document
 createElement $paper title]
dom::document createTextNode $title
 {TclXML Example}
dom::document createTextNode
 [dom::document createElement
$paper abstract] {This is an
example of a XML document instance}
set in [dom::document
 createElement $paper introduction]
dom::document createTextNode $in
 {XML uses the }
set acronym [dom::document
 createElement $in acronym]
dom::document createTextNode
 [dom::document createElement
 $acronym short] DTD
dom::document createTextnode
 [dom::document createElement
 $acronym full] {Document Type
 Definition}
dom::document createTextNode $in
 {to define classes of documents.
 Such a document is known as a }
dom::document createTextNode
 [dom::document createElement
 $in definition] {document
 instance}
dom::document createTextNode $in .

# Add an ID attribute to <abstract>

# Search for the element
foreach child [dom::node children
 $paper] {
 if {[dom::node cget -nodeType] ==
"element"} {
   if {[dom::node cget -nodeName]
== "abstract"} {
    dom::element configure -
attributes {ID abc123}
     break
   }
  }
}

# Write out the XML document

set ch [open example.xml w]

# No explicit serialisation
puts $ch $docRoot

close $ch
```

TclXML

TclXML is a Tcl package which provides the facilities needed by a Tcl script to parse XML documents, traverse and manipulate their structures and to generate XML documents. The package provides an implementation of TclDOM (see below) for accessing and manipulating the document structure. It also provides a framework for the various components needed to parse and generate documents, allowing different implementations to be used together in a seamless fashion.

Applications wishing to use TclXML will have different requirements, both in terms of functionality and in terms of performance. Some applications may require an event-based parser, others may require a tree-based representation. An application may only need to check that a document is well-formed, whereas another may need to validate its documents. Figure 1 outlines the various requirements needed.

Application Requirement	Performance	Non-Validating	Validating
Event-based	Critical	TclExpat	Java Parser
	Non-Critical	TclXML	TclXML + Validator
Tree-based	Critical	TclExpat + TclXML Tree Builder	Java Parser
	Non-Critical	TclXML Tree Builder	TclXML + Tree Builder + Validator

Figure 1: XML Processing Requirements of Applications

At present TclXML provides two non-validating XML parsers. A "native" parser written entirely in Tcl and a Tcl interface to James Clark's `expat` [Clark98] parser called TclExpat. Both of these parsers are event based, ie. they produce a stream of "document events", such as the start and end of elements. TclXML also provides a utility to construct a tree based representation of the document, the Tree Builder. This utility uses the event stream produced by either of the parsers to construct the document tree. Finally, TclXML will provide a document validator. The validator will include a DTD parser and will check the complete tree representation of a document for validity according to the document's DTD.

All of these facilities have been exposed to the application developer, since the different parts may be useful for different applications. Some application may process a XML document as a stream, in which an event-based parser is most useful. Other applications may need to traverse the document structure, in which case a tree-based parser is required. Also, the modular construction of the TclXML toolkit, makes it a simple matter to replace components of it with packages written in other languages. TclXML already includes expat, which is written in C, as an example.

An event-based parser which is used within the TclXML framework is configured to issue calls to the TclXML Tree Builder, which then issues calls to the TclDOM module to construct the document tree. A tree-based parser would make calls directly to the TclDOM module. The Tcl application can call the same commands to construct a document in-memory itself.

Compliant Parsers

As can be seen from Figure 2, TclXML aims to allow any compliant parser to be used as a component of the framework. An example might be to replace the built-in Tcl parser with a SAXDOM-compliant parser written in Java. Once a driver is written for the SAXDOM interface, any parser written using that interface can be used with TclXML.

TclExpat

An Tcl extension, called `TclExpat`, has been created to provide an interface to James Clark's `expat` XML parser. This extension has been written for Tcl version 8.0 and 8.1 (alpha2) as a dynamically loadable library. The extension has been tested on Linux, Solaris and HP/UX. Compiling the extension on other platforms supported by Tcl should present no difficulties. An `expat` parser is created using the `expat` command. When a parser is created a corresponding Tcl command is registered to access the parser, in the usual Tcl fashion. `expat` allows callbacks to be registered for various "document events" such as the start of an element, the end of an element, character data, processing instructions, and so on. The purpose of `TclExpat` is to allow Tcl scripts to be invoked as these callbacks. An application may configure a parser with various options to set the callback scripts. Certain callback scripts have arguments appended to them before evaluation, such as the name of an element and its attribute list for the start of an element.

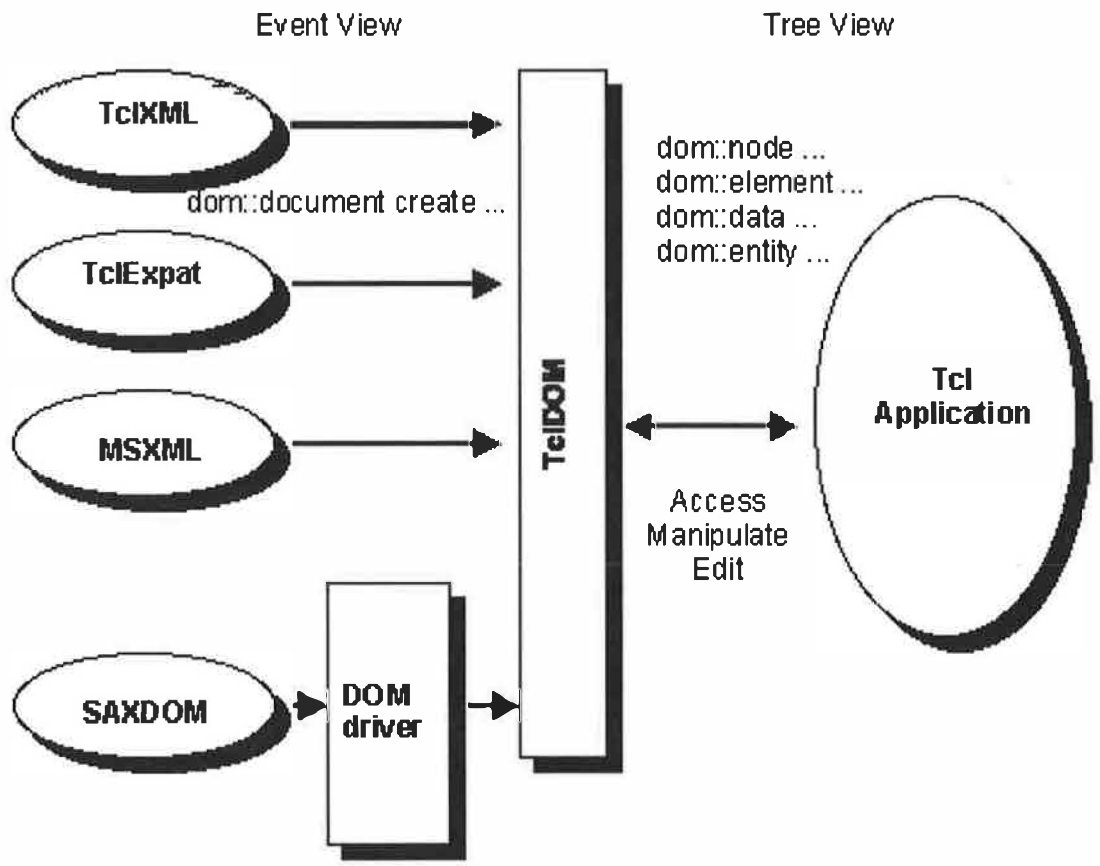


Figure 2: TclXML Modular Structure

The main callback options for the `TclExpat` parser are as follows:

`-elementstartcommand name attributes`

Invoked when an element starts. Arguments are the name of the element and its attribute list, given as a name-value paired list.

`-elementendcommand name`

Invoked when an element ends. The name of the element is appended.

`-datacommand data`

Invoked when character data is encountered. The data is appended.

`-processinginstructioncommand name data`

Invoked when a processing instruction is encountered. Arguments are the name of the processing instruction and the instruction's data.

`-defaultcommand data`

This callback is invoked when data is encountered for which there are no other handlers registered.

In addition to supporting straightforward callback scripts, `TclExpat` also allows these scripts to use the `continue` and `break` commands to alter how the document is processed. If the callback script returns a `TCL_CONTINUE` code then callback invocation is suspended until the currently open element is closed. A return code of `TCL_BREAK` causes all further callback invocation to be skipped. If a callback script returns an error condition all further callbacks are skipped and the parser also returns an error condition. `TclExpat` handles aborting the processing of document callbacks even though `expat` itself does not support this feature.

An example of using `TclExpat` is as follows:

```
package require expat

set parser [expat xmlParser]
$parser conf -elementstartcommand
```

```
  CountElements
$parser conf
  -processinginstructioncommand PI

proc CountElements args {
    incr ::count
}

proc PI {name args} {
    if {[regexp
{break|continue|error} $name]} {
        return -code $name "$name
due to processing instruction"
    }
}

set result {}

set count 0
$parser parse {<?xml
version="1.0"?>
<!DOCTYPE Document SYSTEM
"Document.dtd">
<Document>
<Visible>This element is
processed</Visible>
<Interrupted>This element is
skipped <?break?>
<NotCounted/>
</Interrupted><NotCounted/>
</Document>
}
lappend result $count

set count 0
$parser reset
$parser parser {<?xml
version="1.0"?>
<!DOCTYPE Document SYSTEM
"Document.dtd">
<Document>
<Visible>This element is
processed</Visible>
<Interrupted>This element is
skipped <?continue?>
<NotCounted/>
</Interrupted><Counted/>
</Document>
}
lappend result $count

puts $result
```

The output of this script would be `3 4`.

Document Generation

When data stored in an internal data structure is converted to a XML representation, the data is said to be serialised. In this way, a XML document may be generated by a program.

It is proposed that TclDOM-compliant implementations store their documents as the internal representation of a Tcl Object. When a Tcl script requires such an object in a string format, the object will serialise the document, thus generating a XML document. This process may apply to an entire document, a document fragment or an element hierarchy. After serialisation the document is just a plain Unicode string and the application may write it to a file, a network channel and so on.

Performance

Tests have been conducted to investigate the performance of the `TclExpat` extension and the all-Tcl TclXML parser. The `expat` XML parser source code distribution includes a well-formedness checker. The output of this program is a normalised version of the XML input file. A Tcl version of this program, using either the `TclExpat` extension or TclXML parser, which accepts the same inputs and produces the same output, was written for the purpose of running comparison tests. Figure 3 shows the performance comparison.

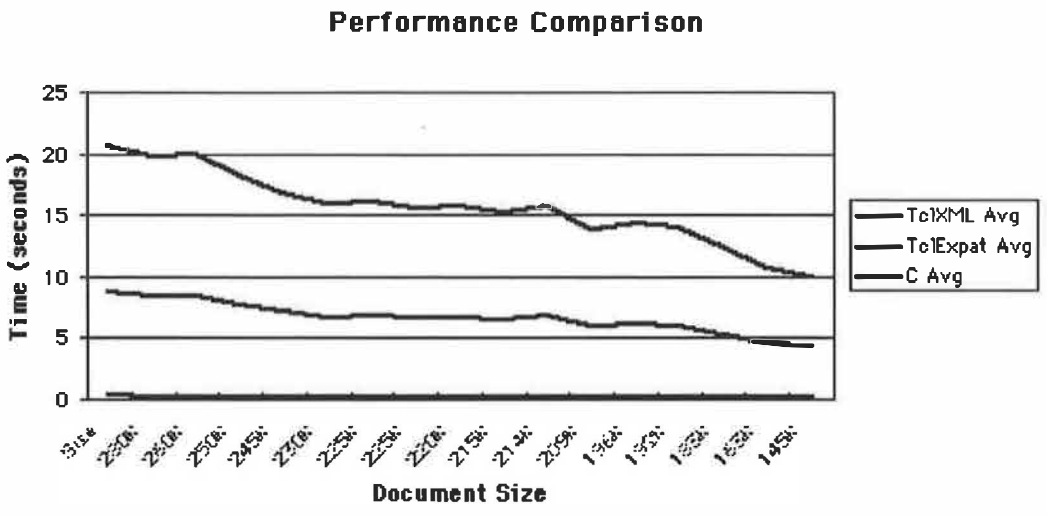


Figure 3: Performance Comparison of C Program with TclExpat and TclXML.

The timing data was calculated using a version of `TclExpat` and TclXML compiled against Tcl version 8.0, which does not support Unicode. Tcl version 8.1 alpha2 was not used because its regular expression engine is known to have poor performance as it has not yet been performance optimised. The tests were conducted on a Sun UltraSPARC 2.

The tests indicate that the Tcl script using the `TclExpat` extension is approximately 10 times slower than the equivalent C code, and the full Tcl version is 20 times slower. Given that the only difference between the two versions of the Tcl program are that the all-Tcl version must also parse the XML document in Tcl, it would appear that the performance loss in the `TclExpat` version is due to the task of writing the normalised XML document. A 10 to 20 times performance loss for Tcl scripts when compared to C code is acceptable. From this we can conclude that the `TclExpat` extension adds minimal overhead to `expat` for parsing XML documents. Further testing on more complex applications will be required to draw conclusions on the feasibility and/or efficiency of using Tcl to process XML documents, but this comparison test is encouraging.

An important point to note is that the C program included with the `expat` distribution is 653 lines of code. The Tcl script developed using the `TclExpat` extension or TclXML package is only 89 lines of code. This indicates that writing document processing applications is significantly more productive using Tcl.

Application Examples

Plume's XML support and TclXML are being used in two applications. Firstly, Steve Ball has written a book, titled "Web Tcl Complete" [Ball98B]. The publisher, McGraw-Hill, required the manuscript to be delivered as plain, double-spaced text, along with code and script examples to be included on a CD-ROM. However, it was also desirable to produce drafts of the chapters for the Web.

To meet all of these requirements the book was written in XML, with markup used to identify code examples along with other structural features. Two simple scripts were written, one to translate the documents into HTML and another to output plain text while at the same time extracting the code examples. The tree-based document representation allowed these scripts to be written quickly and easily. These scripts used the Plume XML support library, which provided some valuable experience to guide the development of the TclDOM specification.

Another application is an Intranet development for the Department of Foreign Affairs and Trade of the Australian Federal Government. For this project it was decided to store corporate documents in XML format, to allow rich structuring of the information for improved searching and vendor- and platform-independence of the data. To support the current generation of Web browsers, the XML documents are translated into HTML on-the-fly at the time of document delivery. If the translation required is straightforward an event-based parser is used. An event-based parser allows large documents to be delivered incrementally, thus reducing the latency of document delivery and improving the perceived responsive of the system to the user. More complex translations use the tree-based representation of the document, which allows advanced formatting depending on the content of elements. These technologies are augmented with Tcl microscripting, using the Tcl Web Server [Tclhttpd], to provide a highly dynamic Web site.

Conclusion

An Application Programming Interface standard is being developed to provide a Tcl language binding for the DOM, known as TclDOM. TclDOM provides the means to create XML documents, as well as access and traverse their document structures. DOM supports both XML and HTML, and so too will TclDOM. A single interface to both standards will simplify the development of Tcl applications which wish to process both XML and HTML documents.

A Tcl package has been developed, called TclXML, which provides a sample implementation of the TclDOM. TclXML includes a non-validating, event-based XML parser written in Tcl, as well as a Tcl interface to the `expat` parser. Modules are, or will be, provided to create the document tree, validate documents and produce XML or HTML output. TclXML is being designed to take advantage of the significant number of Java packages that are available for processing XML documents.

XML, DOM and other related standards are, or will be, Recommendations of the World Wide Web Consortium which will have a major impact on the World Wide Web, as well as making inroads into application areas which currently use SGML. Many network protocols and document formats are proposing to use XML as their data syntax, such as CDF, RDF and PGML to name but a few. This means that it is very important for the Tcl language to be able to efficiently process XML documents, in order to be able to support the development of applications which make use of XML. Tcl 8.1 has the advantage that it supports Unicode, though Tcl 8.0 may also be used for western character sets.

Given the increasing importance of XML for a wide range of application development it is the author's opinion that XML support of some form must be included in the core Tcl distribution. TclXML is being distributed under a license that permits it to be added to Tcl. It is interesting to note that Perl is also undergoing development to include Unicode and XML support [Perl]. Tcl has a significant advantage in that it is already Unicode-enabled, and so has a lead on Perl development.

References

[TCLXML]
Zveno Pty Ltd. *Tcl Support For XML.*
http://www.zveno.com/zm.cgi/in-tclxml/

[XML]
Tim Bray, et al. *Extensible Markup Language.*
World Wide Web Consortium.
http://www.w3c.org/TR/

[W3C]
World Wide Web Consortium. http://www.w3c.org/

[DOM]
Lauren Wood, et al. *Document Object Model.*
World Wide Web Consortium.
http://www.w3c.org/TR/

[CSS]
Hakon Lie, et al. *Cascading Style Sheets.*
http://www.w3c.org/Style/

[Ball98A]
Steve Ball. *The Plume WWW Browser.*
http://plume.browser.org/

[Ball98B]
Steve Ball. *Web Tcl Complete*
McGraw-Hill, New York.
http://www.zveno.com/zm.cgi/in-wtc/

[Uhler95]
Uhler, S. *html_library.*
ftp://ftp.smli.com/research/tcl/html_library.tar.gz

[Clark98]
James Clark. *Expat XML Library.*
http://www.jclark.com/xml/

[English96]
Joe English. *CoST.* http://www.art.com/cost/

[PM-R96]
Peter Murray-Rust. *CoSTWish.*
http://www.venus.co.uk/omf/costwish/

[Nicol98]
Gavin Nicol, Inso Corporation. mailto:gtn@inso.com

[Tclhttpd]
Brent Welch, Scriptics Corporation.
http://www.scriptics.com/

[Perl]
Larry Wall. http://perl.oreilly.com/

Creating High Performance Web Applications using Tcl, Display Templates, XML, and Database Content

Alex Shah
Technical Director
Binary Evolution, Inc.
ashah@binevolve.com

Tony Darugar
President
Binary Evolution, Inc.
tdarugar@binevolve.com

Abstract

We describe an online system that provides a framework for the rapid creation of high performance, database driven web sites based on content from XML files. The software that "glues" the content to the presentation is written in Tcl. The proposed architecture uses a pool of persistent Tcl engines to substantially improve performance and robustness as compared to traditional server-side programming techniques.

Introduction

Today's online applications demand more from web technology than C-based CGI programming can provide. A web site is a living document: the content, the presentation, and the software that drives that presentation need to change often. To meet the day to day requirements of a dynamic web site, a developer must use tools and technology that maximize flexibility and minimize development time.

We will describe an online system that provides a framework for the creation of high performance, database driven dynamic web sites. Simple HTML templates that can be manipulated in WYSIWYG editors will be used for display of the content. Content will be stored in XML documents or within a database, allowing publishers to update the site easily. The software that glues the database and XML content to the display templates will be written with one of the most stable, well supported scripting technologies available: Tcl. The Tcl language was chosen because it is non-proprietary, platform and operating system independent and is familiar to most web developers. Superior web performance will be achieved through the use of Tcl engines that maintain persistent database connections and communicate directly with the web server via the ISAPI or NSAPI protocols.

Performance Issues

Traditional server-side programming has relied heavily on the Common Gateway Interface (CGI) standard, first implemented in the NCSA server. The CGI protocol is fairly easy to understand and allows developers to write server-side applications in the language of their choice. Since applications run in separate processes, CGI based applications can crash without bringing down the entire web server. Nearly every web server, regardless of platform, has implemented the CGI standard.

CGI has some significant performance drawbacks. The operating system overhead associated with starting a new process for each request often utilizes more resources than the script itself. Under load, incoming requests are scattered by the scheduler: some requests are processed immediately; others may wait for hundreds of seconds. When content is stored in a database, the additional overhead from opening and closing the connection often results in huge lags, even for the simplest transaction.

In response to the performance problems associated with CGI, several vendors have developed high performance, proprietary APIs for their servers. The most notable are Netscape's NSAPI, Microsoft's ISAPI and Apache's server API. Applications compiled using one of these proprietary server APIs are faster than CGI programs. By running an application within the server process, the overhead of starting up and initializing a new process for each request is removed. Requests are handled in the order received and are not subject to rearrangement by the operating system's process scheduler. Database connections can be opened once, and reused.

Unfortunately, server APIs are missing many of CGI's benefits. Server APIs are difficult to learn and force the developer to use C or C++ to develop their application. Since the application is loaded into the server process, a bug in the application can bring down the entire server. Once written, an application cannot be easily ported to a new server or platform without rewriting the code.

Several solutions try to combine the ease of use of CGI and the performance of server APIs. One approach is the creation of an entirely new programming language and API for the sole purpose of server programming. Cold Fusion is an example of this approach. While easier to use than server APIs, Cold Fusion has several obvious disadvantages, such as the need to learn a new, proprietary language. Cold Fusion users do not have access to a large, established development community and must rely on a single vendor for all advances and extensions to the language.

Unlike Cold Fusion, NeoWebScript and others use established open languages such as Tcl that are already familiar to many developers and have a large user and development community [Lehen]. NeoWebScript takes the approach of adding a Tcl interpreter to the Apache server process and allows the web developer to embed Tcl code within their HTML pages. Since a new Tcl process is not created for each request, NeoWebScript's performance is substantially better than CGI. Despite NeoWebScript's ease of use and performance, a few drawbacks do exist. The approach of embedding the Tcl interpreter directly into the http process only works for non-multithreaded servers such as Apache. Existing CGI-Tcl scripts must be rewritten to conform to NeoWebScript's programming paradigm.

Another approach is to improve and extend the CGI protocol by removing the overhead of starting a new process for each new request. FastCGI implements this approach by introducing an accept loop, whereby the script accepts an incoming request, processes it, and goes back the accepting state [FastCGI]. Many of the benefits of CGI are retained by this approach, including being language independent and protecting the http server process by executing the scripts in a separate process. However, FastCGI's improved performance comes at the expense of greater effort on the part of the developer . Since the scripts are required to be persistent, they must be memory and resource leak free. The cleanup automatically done by the operating system when using CGI (by restarting the process for each request) must now be handled by the developer. FastCGI encourages the creation of large, monolithic programs, as opposed to small, modular scripts in order to have less processes to manage [FastCGI2].

Ideally, an improved CGI would increase performance as well as ease server-side programming, rather than complicate it. A successful CGI adaptation would take the benefits of the existing CGI protocol, combine them with the performance advantages of server APIs, be easy to use, and be based on open languages.

High Performance Architecture

The architecture we will use for our sample application uses Binary Evolution's *VelociGen*™ [VelociGen]. The *VelociGen for Tcl*™ (VET) combines the performance associated with server APIs with the benefits of CGI (see Figure 1).

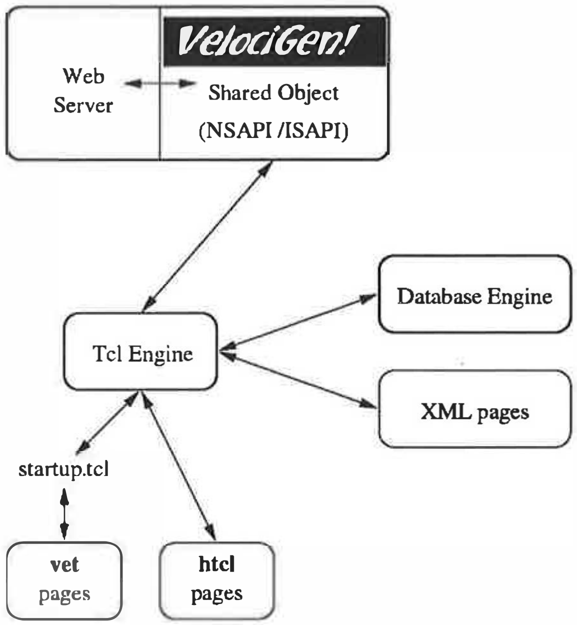


Figure 1. *VelociGen*™ Architecture

In VET's default configuration, requests for files that end with **.vet** or **.htcl** are processed by the *VelociGen*™ interface. VET can be configured to run scripts with different file extensions, or from a specific directory, for example **/cgi-bin**.

When a Tcl request is received by the http server, the *VelociGen*™ shared object searches for an available Tcl engine to handle the request. If no Tcl engines are available and the user defined maximum number of Tcl processes has not been exceeded, a new Tcl engine is spawned. The request is queued when all of the Tcl engines are busy processing prior requests.

Upon startup, the Tcl engine creates a master interpreter which remains in memory until the Tcl engine is exited. The master interpreter evaluates a file called **persistent.tcl** which contains user generated Tcl code that will be shared by all requests. Code for creating persistent socket or database connections should be added to the **persistent.tcl** file. Since code evaluated by the master interpreter is pre-compiled and cached, maximum Tcl performance can be obtained by moving procedures from your Tcl script into **persistent.tcl**.

After executing **persistent.tcl**, the Tcl engine waits for a request to arrive from the parent process. The *VelociGen*™ shared object will pass the filename for the script, CGI variables, and POST data to the Tcl engine. After this information is received, the Tcl engine will evaluate a file called **slave.tcl**. The purpose of **slave.tcl** is to create a new slave interpreter and alias any necessary procedures from the master interpreter. By modifying **slave.tcl** the developer can choose whether the Tcl script should be run using a safe or unrestricted interpreter.

If the script to be run is mostly HTML with some embedded Tcl code (**.vet**), the **startup.tcl** file will be evaluated next. The **startup.tcl** file contains Tcl code to simplify CGI programming. For example, it stores the query string in a Tcl associative array called QUERY and cookies into the COOKIE associative array. It also provides support for multipart form data, http file uploads and miscellaneous procedures for common web programming tasks. Depending on user preference, the **startup.tcl** file can be easily replaced with another CGI Tcl library such as the cgi.tcl package written by Don Libes [Libes]. After executing **startup.tcl**, the **.vet** file is processed as follows: HTML is passed onto the web server without further processing by the Tcl engine. Tcl code defined within user specified tags, for example: <tcl>...</tcl> is evaluated by the slave interpreter; standard output is passed onto the web server.

Scripts ending with **.htcl** run in VET's CGI compatible mode. In this mode, **startup.tcl** is not evaluated. CGI variables are copied to Tcl's *env* array. POST data is placed on *stdin*; data written to *stdout* will be communicated to the web server and eventually reach the client browser.

In the event that a Tcl script crashes the interpreter, only the current request is affected. Other Tcl engines, and the server http process continue to run without incident. The *VelociGen*™ shared object can detect when a Tcl engine has crashed, display an appropriate error message on the client browser, and restart the engine when the next Tcl request is received. *VelociGen*™ can also be configured to forcefully terminate a script after a user defined number of seconds.

After script termination, the slave interpreter is destroyed, freeing variables, file handles, and other resources used in the processing of the **.vet** or **.htcl** script. The master interpreter is not cleared out: Tcl code, variables, and socket or database connections defined in **persistent.tcl** remain cached and pre-compiled in the Tcl engine. The Tcl engine now waits for another Tcl request to arrive. The connection between the http server and Tcl engine is never closed.

Comparison of *VelociGen*™ to Other Server-Side Programming Solutions

Like CGI and FastCGI, applications written with VET run in separate, isolated processes. Scripts submitted to the Tcl engine for processing can crash, block on IO, or go into infinite loops, without affecting the web server. Only a small amount of well tested, thread-safe code for managing and communicating with the Tcl engines needs to be loaded into the web server.

Unlike CGI, the Tcl process or engine remains persistent and does not exit after each request, thus eliminating substantial operating system overhead. By keeping the Tcl engine in memory, persistent connections can be made to the database, further increasing performance. VET requests are less susceptible to rearrangement by the operating system process scheduler. Since requests are handled in the order that they are received, response time stays consistent, even under heavy load.

By separating the web server and Tcl into separate processes, the potential exists for running the Tcl engines on remote machines. The architecture provides scalability: request handling and server performance can be easily increased by distributing Tcl processing over several machines.

Unlike FastCGI, VET is capable of handling existing CGI-Tcl scripts without modification. Developers can continue to code small scripts without performance loss. Freeing of system resources is handled by VET rather than the developer. Several requests to the same Tcl script are processed simultaneously, rather than serialized in an accept loop.

Rather than taking Cold Fusion's approach of creating a entirely new language for server-side programming, VET shares the approach taken by NeoWebScript. The VET interface is based on the well established and proven Tcl language. VET pages can also parse and process Tcl embedded within HTML pages. The embedded Tcl code is placed within user defined tags, for example: <tcl>,

</tcl>, <%, %>, [[,]]. Unlike NeoWebScript, VET is backward compatible with CGI scripts written in Tcl. By separating the Tcl interpreter from the http server process, thread-safe *VelociGen*™ technology supports multithreaded servers such as Netscape Enterprise, Microsoft IIS, and O'Reilly Web Site Pro, as well as Apache.

The VET interface is not dependent on a particular platform or server architecture. Any web server that provides a high performance interface compatible with Netscape's NSAPI, Microsoft's ISAPI, or Apache's server API will work with VET. Since Tcl has been designed to run on nearly any platform, the VET interface allows developers to write an application once and then deploy on nearly any operating system, platform, or web server.

Why use Tcl?

Scripting languages such as Tcl represent a very different style of programming than system programming languages such as C or Java™. Scripting languages are designed to achieve a higher level of programming and more rapid application development than system programming languages. Scripting languages also provide a simplified environment for performing online tasks. Content can be pulled from databases or XML and manipulated using regular expressions or other built-in string functions without having to build data structures and algorithms from scratch.

The Tcl language was chosen as a base for our system because it has only a few fundamental constructs and relatively little syntax, which makes it easy to learn. The Tcl syntax is meant to be simple and flexible. Developers can easily add commands and functionality to Tcl using C or C++ code when needed. As John Ousterhout, the creator of Tcl puts it, Tcl is designed for "gluing" applications together.

Support and extensions for Tcl are widely available, making it a very attractive language for online development. In addition to the text manipulation functions that come with Tcl, a powerful regular expression library can be installed into the interpreter. Support for nearly any database is available for Tcl. Steve Ball has recently made an XML extension available as well [Ball]. Extensions also exist for creating and manipulating graphs and GIF images.

Tcl 8.0 also comes with a byte compiler. VET's ability to pre-compile and cache Tcl code is ideal in a web environment where scripts are run over and over. By using pre-compiled Tcl, the overhead of interpreting Tcl source code is removed. Since most Tcl routines map to compiled C, pre-compiled Tcl can execute at speeds that come very close to the performance of server APIs.

With respect to online publishing, Tcl scripts are especially powerful. A developer can quickly and easily update a script and see the changes on the next server request, without having to recompile the entire application. Tcl allows editors, graphics artists and developers to update the content, the presentation of that content, and the logic driving that presentation on a daily basis, while still maintaining high performance and flexibility.

Online Recipe Application based on XML and Tcl

A vast amount of documentation and information is available on the Internet, with more being added at an incredible pace every day. Most of this information is stored in HTML, the Web's ubiquitous formatting language. This has been a very successful model both because of the ease of authoring HTML documents and because of the relative ease of displaying them.

While HTML is effective as a formatting and display language, it does not allow the author of the document to store structural or fielded data within the document. In fact, the only information maintained within the document is how to display it. More and more, the need for structured, fielded documents is becoming obvious.

To demonstrate this, imagine all of the different cooking recipes available on the Internet. Each has its own look, order for placement of sections and wording for the headings. If you wanted to collect these disparate recipes into a single collection with a uniform look, you would have to expend a large amount of energy figuring out how each recipe is laid out and converting it to your own look. Or, imagine if you wanted to place the ingredient list of each recipe into a central database, so that you could search for recipes that used a particular combination of ingredients. Since each recipe lists its ingredients in a different way, a large amount of effort would be needed. These data management issues also apply to collections of movie reviews and stock portfolio information.

XML can be used to remedy this situation. An XML tagged document resembles HTML, although the meaning of those tags are defined by the author of the document [XMLRec]. Unlike HTML, XML tags add meta information to the document rather than determine how the document will be displayed. By separating the display characteristics of the document from the content, XML allows one to change the presentation without having to modify content spread across hundreds or thousands of documents. Also, since the tags are structured and can be nested, meta-data can be stored within the document itself. For example, a recipe document could have the tag <ingredient>carrot</ingredient> to indicate that carrots are used in that recipe, making it very obvious what each ingredient is. The following listing shows a partial recipe for carrot cake in XML:

```
<recipe course="dessert" calories="325">
  <name>Carrot Cake</name>
  <description>
    A favorite classic. Stays moist and freezes well.
  </description>
  <note status="credit">Betty Crocker's Cookbook</note>
  <ingredient-list yields="16">
    <ingredient quantity="1 1/2 cups">sugar</ingredient>
    ...
    <ingredient quantity="3 cups">shredded carrots</ingredient>
    <ingredient quantity="1 cup">coarsely chopped nuts</ingredient>
  </ingredient-list>
  <preparation preptime="about 1 1/2 hours">
    <step>Heat oven to 350 degrees</step>
    <step>Grease and flour rectangular pan, 13 x 9 x 2 inches</step>
    ...
   <step>Frost with cream cheese frosting if desired</step>
  </preparation>
</recipe>
```

Listing 1. Carrot Cake Recipe in XML

Using XML important information about the structure of a document can be placed within the document itself.

The following example demonstrates how *VelociGen for Tcl*™ can be used in conjunction with an XML parser to display recipes stored in XML format.

Since XML tags are guaranteed to be balanced, valid XML documents can be parsed and placed into a tree data structure. For our example, we use Steve Ball's XML parser for Tcl [Ball] which transforms the XML document into a tree structure using Tcl lists. Each node in the tree is made up of four elements:

node: type tag attribute content

The *type* element is set to one of two values: 'parse:elem' or 'parse:text'. When *type* is 'parse:elem', *tag* is set to the name of the XML tag, *attribute* is set to a list of attributes placed within the XML tag, and *content* is set to one or more *nodes* found between the start and end tags. When *type* is 'parse:text', *tag* is set to the text found between the start and end tags, *attribute* and *content* are set to empty Tcl lists: '{}'.

A simple approach to giving XML presentation is to simply map the XML tags to appropriate HTML tags. Listing 2 shows this approach. The *tag_map* defines these simple mappings - one for the start of each tag, and one for the end. As the tree is recursively traversed, using the *process* function, each tag is handled as follows: the HTML code defined by the *tag_map* is displayed, followed by the content of the tag, followed by the end HTML code defined by the *tag_map*.

```
# ----------------------------------------
# The tag map: Each item consists of the tag name, the HTML to output at the beginning
# of processing the tag, and the HTML to output at the end of processing the tag.
set tag_map {
{RECIPE ""      "" }
{DESCRIPTION    {<blockquote>}  {</blockquote>} }
{NAME   {<center><h2>}  {</h2></center>} }
{NOTE   {<!--}  {-->} }
{INGREDIENT-LIST        {<p><h3>Ingredients</h3><ul>}   {</ul>} }
{INGREDIENT     {<li>} "" }
{PREPARATION    {<p><h3>Preparation</h3><ol>}   {</ol>} }
{STEP   {<li>}  "" }
}
```

Listing 2. XML to HTML conversion tag_map

The XML tags are processed using a simple recursive traversal of the parse tree:

```
# Output start HTML, XML tag contents, and
# end HTML as defined in tag_map for the
# XML tag "$tag"

process_tag start $tag
process $content
process_tag end $tag
```

Using this approach XML documents can be converted to HTML quickly and easily. However, Listing 2 does not allow for the handling of tag attributes - they are simply ignored. To deal with the more complicated aspects of XML we need a more powerful mapping.

The more powerful approach involves defining Tcl procedures to handle the XML tags. Rather than using text substitution, each XML tag becomes a Tcl procedure, taking the *attributes* and *content* as arguments. This approach provides the full power of Tcl for handling the XML tags.

Listing 3 shows this approach. *tag_map* is now a namespace instead of a Tcl list. Each procedure in this namespace corresponds to an XML tag and defines what is to be done with that tag.

```
namespace eval tag_map {

proc DESCRIPTION {attr content} {
    puts "<blockquote>"
    process $content
    puts "</blockquote>"
}

proc NOTE {attr content} {
    set value     [lindex $attr 1]
    switch $value {
        credit {
           puts "<br>From: <i><b>" ; process $content ; puts "</b></i><br>"
        }
        default { }
    }
}

proc INGREDIENT-LIST {attr content} {
    puts "<table border=0 bgcolor=yellow width=100%><tr><td>"
    puts "<h3>Ingredients:</h3><ul>"
    process $content
    puts "</ul></td></tr></table>"
}

proc INGREDIENT {attr content} {
    puts "<li><b>"
    process $content
    puts "</b><i>[lindex $attr 1]</i><br>"
}

...
# End tag_map namespace
}
```

Listing 3. XML to HTML conversion namespace.

A typical procedure inspects and processes the tag attributes, outputs some HTML code, calls *process* to recursively handle the contents of the tag, and outputs the ending HTML code. The call to *process* is needed in order to reach the current tag's child *nodes*; omitting it allows you to suppress the display of the tag's contents, including its subtrees. For example, by removing the *process* call from the *INGREDIENT-LIST* procedure, the listing of ingredients can be suppressed.

Using this system the documents can be written and maintained in XML, providing all of the benefits discussed above, and displayed in HTML, allowing the system to be deployed today, using existing web technologies. Further benefits of authoring and storing the documents can be seen in the following section, where the XML document is very easily converted to SQL and stored in a database.

Online Recipe Application based on Databases and Tcl

XML provides structure to a document, but does not provide a mechanism to search and query the content. For such purposes, it is often useful to store content within a database. Our next example will demonstrate how recipe information can be stored in a database and then retrieved and displayed in HTML using Tcl.

Listing 4 shows the schema for storing the recipe example in a SQL database. The recipe is stored in three tables: *recipe*, which stores the meta level information about the recipe; *ingredients*, which contains the list of ingredients; and *prep*, which contains the preparation steps.

Listing 3, which defines the mapping from XML to HTML, can be modified to map from XML to SQL statements, allowing us to store our document in a SQL database. Listing 5 shows the necessary modifications. Notes are ignored simply by not processing their contents.

```
create table recipe (
        id              int,
        course          char(1),
        calories        int,
        name            char(30),
        description     char(254)
);

create table ingredients (
        rec_id          int,
        quantity        char(20),
        name            char(20)
);

create table prep (
        rec_id          int,
        step_num        int,
        description     char(254)
);
```

Listing 4. Database Schema

```
namespace eval tag_map {

proc RECIPE {attr content} {
    puts
       "insert into recipe (id) values (1);"
    process $content
}

proc NAME {attr content} {
    puts "update recipe set name='[process $content]';"
}

proc DESCRIPTION {attr content} {
    puts "update recipe set description='[process $content]';"
}

proc NOTE {attr content} {
    # Ignored
    return
}

proc INGREDIENT-LIST {attr content} {
    process $content
}

proc INGREDIENT {attr content} {
    puts "insert into ingredients values (1, '[lindex $attr 1]', '[process $content]');"
}
...
# End tag_map namespace
}
```

Listing 5. XML to SQL tagmap namespace

To create SQL output instead of HTML, only the mappings need to be redefined. This demonstrates the flexibility and power of XML. By adding structure to a document without including presentation information, XML allows easy conversion of the document to other formats, including HTML and SQL.

Listing 6 shows a code segment for displaying the recipe, drawing data from the database. The necessary data is retrieved from the database using SQL statements and made into HTML via an HTML template. The HTML template can be manipulated using an HTML editor, allowing site designers to modify the look of the page without having to understand the Tcl code. The following listing uses the MySQL database with Tcl-GDBI [Darugar] as the Tcl interface.

```
...
<tcl>
set select "select * from recipe where id=1"
set nrows [sql query $conn $select]
set row     [sql fetchrow $conn]

# get the fields out of the returned row.
set idx 0
foreach field {id course calories name description} {
        set $field [lindex $row $idx]
        incr idx
}
</tcl>

<center><h2><tcl>puts $name</tcl></h2></center>
<blockquote>
<tcl>puts $description</tcl>
</blockquote>

<!-- Ingredients -------------------------------------- -->
<table border=0 bgcolor=yellow width=100%><tr><td>
<h3>Ingredients:</h3>
<ul>
<tcl>
set select "select * from ingredients where rec_id=1"
set nrows [sql query $conn $select]

while {[set row [sql fetchrow $conn]] != ""} {
        set quantity [lindex $row 1]
        set name     [lindex $row 2]
</tcl>

<li><b><tcl>puts $name</tcl></b><i><tcl>puts $quantity</tcl></i><br>

<tcl>
# end of while loop for ingredients
}
</tcl>

</ul></td></tr></table>
...
```

Listing 6. Displaying content from a database.

In real life situations drawing content from a database or using XML/SGML to serve popular sites has often been avoided because of the performance implications: each request requires accessing a database and formatting the content into HTML, or parsing and translating XML into HTML. Traditional server programming systems such as CGI make this overhead prohibitive, forcing the developer to use static HTML instead.

Performance Results

VET shrinks the performance gap between static HTML and dynamically generated HTML. Figure 2 shows response times for displaying recipes at various user loads using CGI-Tcl scripts, VET driven Tcl scripts, and static HTML. The Tcl scripts used for the performance comparison retrieve and display recipe data from an Oracle database (see Listing 6). For testing purposes, slight modifications were made to Listing 6 in order to use the exact same script under CGI and VET. The script was converted into CGI-Tcl by placing **puts** before each line of HTML and removing the <tcl>, </tcl> tags. We also replaced our MySQL interface [Darugar] with the more widely used Oratcl library [Poindexter]. Additional performance was attained by maintaining persistent database connections in VET. Tests were done on a Sun IPX running Solaris 2.5.1, Netscape Fasttrack Server 2.01, Oracle Workgroup Server 7.3.2.2.0 and *VelociGen for Tcl*™ v1.0c.

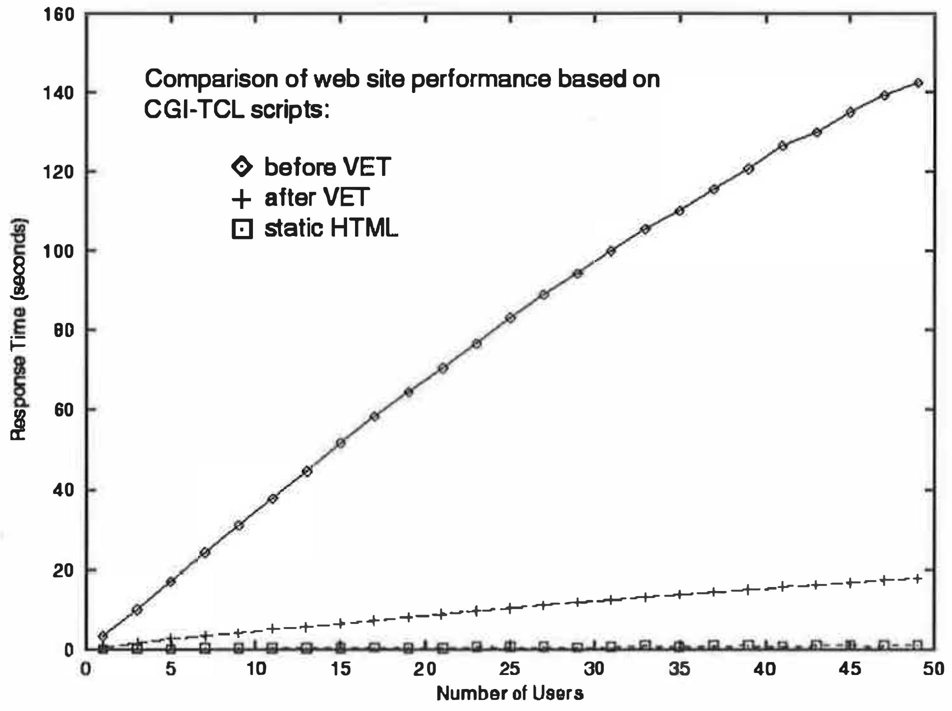


Figure 2. *VelociGen for Tcl*^{TM} vs. CGI Performance

As seen in Figure 2, VET's performance is much better than an equivalent system written with CGI. As the load increases to 50 simultaneous requests, the average response time for the CGI based system jumps to over 140 seconds, which is unacceptable for most web applications. The VET based system handles the load well, taking about 10-20 seconds under stress. Requesting the recipe as a static HTML page does outperform the VET solution slightly, a small price to pay for the ability to search, query your XML data and manipulate the presentation.

Conclusion

Tcl is very well suited for online applications. The language provides a simple syntax and built-in functions for handling many online tasks. It is multiplatform, well tested and supported, and is non-proprietary. It also has the additional benefit of being supported by an entire community of developers rather than a single vendor. This allows users to take advantage of the latest technologies as they appear, well before they can be implemented into proprietary systems. In our simple recipe example, we have demonstrated Tcl's ability to easily load and display content from both databases and XML.

When combined with Binary Evolution's *VelociGen*^{TM}, the use of Tcl for the creation of high performance web sites becomes possible. The proposed system cleanly separates content from presentation, allowing easy manipulation of both the site's look and substance. Rather than waiting 2 to 3 years for widespread XML support in the browser, this solution allows developers to use XML technology today. By tightly integrating database technology, Tcl and HTML templates, our proposed system provides a flexible framework for meeting the day to day needs of today's dynamic web sites.

References

[Ball]	Steve Ball, Steve.Ball@tcltk.anu.edu.au, "TclXML", URL: http://tcltk.anu.edu.au/XML/
[Darugar]	Parand T. Darugar, tdarugar@binevolve.com, "Tcl-GDBI : Tcl MySQL interface", URL: http://www.binevolve.com/~tdarugar/tcl-sql/
[FastCGI]	Open Market Inc., "FastCGI", URL: http://www.fastcgi.com/
[FastCGI2]	Open Market Inc., "FastCGI: A High-Performance Web Server Interface", April 1996, URL: http://www.fastcgi.com/kit/doc/fastcgi-whitepaper/fastcgi.htm
[Lehen]	Karl Lehenbauer, karl@neosoft.com, "NeoWebScript", URL: http://www.neosoft.com/neowebscript/
[Libes]	Don Libes, libes@nist.gov, "cgi.tcl", URL: http://expect.nist.gov/cgi.tcl/
[Libes96]	Don Libes, libes@nist.gov, "Writing CGI scripts in Tcl", Fourth Annual USENIX Tcl/Tk Workshop, 1996, URL: http://www.usenix.org/publications/library/proceedings/tcl96/libes.html, URL: http://www.mel.nist.gov/msidlibrary/summary/9622.html
[Poindexter]	Tom Poindexter, "Oratcl" and "Sybtcl", URL: http://www.nyx.net/~tpoindex/tcl.html#Oratcl URL: http://www.nyx.net/~tpoindex/tcl.html#Sybtcl
[VelociGen]	Binary Evolution, Inc., info@binevolve.com, "VelociGen: Fast, Efficient, and Simple Server Programming with Perl and Tcl", URL: http://www.binevolve.com
[XMLRec]	Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0" W3C Recommendation 10-February-1998, URL: http://www.xml.com/axml/target.html

WebWiseTclTk: A Safe-Tcl/Tk-based Toolkit Enhanced for the World Wide Web

Hemang Lavana Franc Brglez
CBL (Collaborative Benchmarking Lab), Dept. of Computer Science, Box 7550
NC State University, Raleigh, NC 27695, USA
http://www.cbl.ncsu.edu/

Abstract

The `WebWiseTclTk` *toolkit is an enhancement of the existing feature set of Safe-Tcl and Safe-Tk that does not compromise security. The toolkit re-defines the functionality of the auto_load mechanism in Tcl such that it works for packages located anywhere on the World Wide Web. It also re-introduces several commands not available in Safe-Tk such as* `toplevel` *and* `menu` *to provide a much richer feature set of Tk commands. The toolkit is written entirely in Safe-Tcl/Tk and uses the* home policy *for running applications as Tcl-plugins.*
The toolkit supports (1) creation of new Web-based Tcl applications with greatly enhanced functionality, and (2) migration of existing Tcl applications to the Web by merely writing an encapsulation script. We demonstrate the capabilities of the `WebWiseTclTk` *toolkit by readily creating an encapsulation script for Web-based execution of the* Tk Widget Demonstrations, *distributed with the core Tcl/Tk.*

Keywords: plugins, Web browsers, security, scripting, encapsulation, GUI.

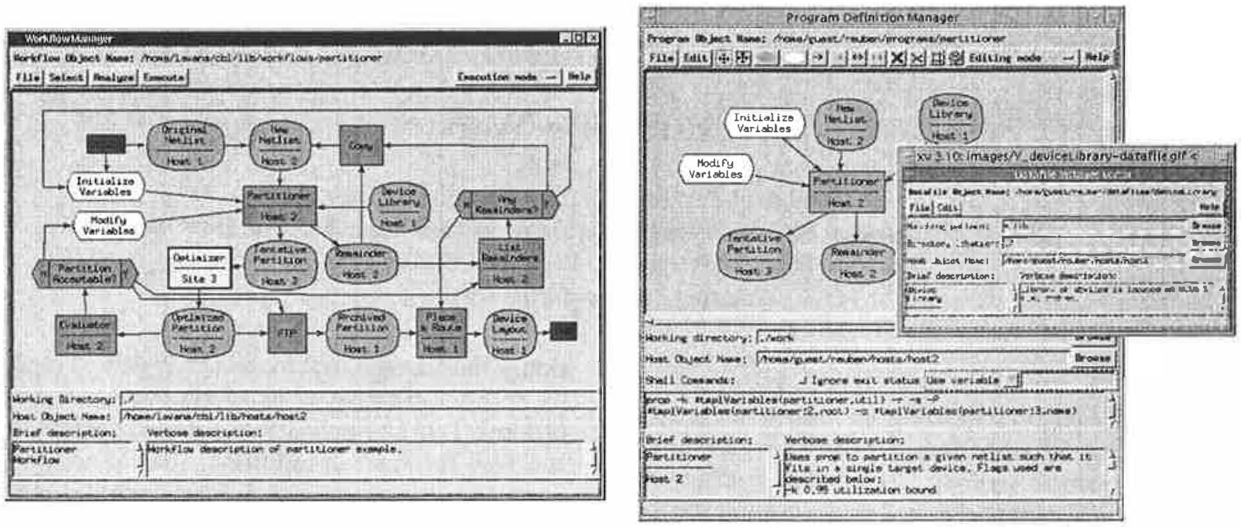
1 Introduction

The last few years have seen an explosive growth of the usage of Tcl (Tool Command Language) [1, 2, 3] and its popularity can be easily gauged by the large number of postings in the Tcl newsgroup `comp.lang.tcl`. Scripting languages such as Tcl are designed for 'gluing' applications and encourage rapid application development as compared to system programming languages, and hence are very important for applications of the future [4]. The emergence of organizations such as the recently formed *Scriptics* [5] and the *Tcl/Tk Consortium* [6], focusing entirely on scripting tools, applications and services, is an example of this trend.
The maturity and robustness of Tcl/Tk provides a new opportunity to support creation and presentation of multimedia content on the WWW. Tcl-plugin [7, 8] is an example of an elegant solution for embedding Tcl/Tk applications for ready access inside the Web browser. In addition, the Tcl-plugin supports an excellent mechanism for security of client hosts using a padded cell approach [9]. The default security policy prohibits Tcl applets (*tclets*) from running other programs, accessing the file system, and creating toplevel windows (including menus), thereby giving the client hosts a high level of confidence when executing *tclets*. However, such restrictions limit the scope of the Tcl applications executed inside a Web-browser.

Our initial experience with Tcl/Tk, predating the phenomenal growth of WWW, was motivated by the need to develop a user-friendly and versatile environment to support user-reconfiguration of complex workflows that execute heterogeneous programs and data for the design of experiments in VLSI CAD. This environment, called `REUBEN` (for reusable and reconfigurable benchmarking environment), was implemented entirely in *Tcl/Tk* [1] and *Expect* [10]. In essence, it provides the user with the ability to create directed dependency graphs as workflows of data, program, decision, and workflow nodes. Data and programs can reside anywhere on the Internet, and execution of all nodes can be scheduled automatically, regardless of the data-dependent cycles in the graph. In its final form, the workflows in `REUBEN` can be multi-cast to several collaborating sites, recorded, and played-back for re-execution. An example of `REUBEN` environment to support a number of distributed and heterogeneous tasks in a VLSI CAD workflow is illustrated in Figure 1. More details are available in [11, 12, 13].

Migration of large applications, such as `REUBEN`, to the Web is not easy if highest level of confidence in terms of security is desired. This is especially true, because toplevel windows and menus are essential in such applications. One solution could be to use `Jacl` [14], an interpreter to run Tcl scripts in a Java environment. Unfortunately, Jacl does not yet contain the entire feature set of Tcl, including

This research was supported by contracts from the Semiconductor Research Corporation (94-DJ-553), SEMATECH (94-DJ-800), and DARPA/ARO (P-3316-EL/DAAH04-94-G-2080 and DAAG55-97-1-0345).



Workflow Object Name: /home/lavana/cbl/lib/workflows/partitioner

Working Directory: ./

Host Object Name: /home/lavana/cbl/lib/hosts/host2

Brief description: Partitioner Workflow

Verbose description: Workflow description of partitioner example.

Program Object Name: /home/guest/reuben/programs/partitioner

Working directory: ./work

Host Object Name: /home/guest/reuben/hosts/host2

Shell Commands: Ignore exit status Use variable

```
prop -k #taplVariables(partitioner,util) -r -s -P
#taplVariables(partitioner:2,root) -c #taplVariables(partitioner:3,name)
```

Brief description: Partitioner Host 2

Verbose description: Uses prop to partition a given netlist such that it fits in a single target device. Flags used are described below:
-k 0.95 utilization bound

xv 3.10: images/Y_deviceLibrary-datafile.gif

Brief description: Device Library

Fig. 1. REUBEN environment consisting of several windows.

namespaces and Tk. The WebWiseTclTk toolkit provides an easy solution for the migration of existing Tcl applications to the Web. Minimal changes are required in the original application. Our approach uses an encapsulating script to call the main script of the original application.

This paper is organized into the following sections: (2) motivation; (3) WebWiseTclTk architecture; (4) implementation of WebWiseTk; (5) implementation of WebWiseTcl; (6) user's guide; (7) programmer's guide; (8) software status and availability; and (9) conclusions.

2 Motivation

A large application, written in Tcl, typically consists of a short main script and a library of support scripts. Applications start up quickly by invoking the main script. As new features are accessed, the code that implements them is loaded automatically, using the auto_load mechanism available in Tcl. A complex environment such as REUBEN, described earlier and illustrated in Figure 1, requires that a number of windows be created during its runtime.

The Tcl-plugin, based on Safe-Tcl, restricts running such large applications inside a Web-browser. A few of these restrictions are listed below:

- Auto_load scheme fails, unless the application package is installed on the client host. Another alternative is to merge all the scripts in the application into a single script which can be downloaded as a *tclet*.
- Applications are restricted to a single window since the command toplevel is not available in Safe-Tk and new windows cannot be created.
- Menu widgets are also disabled in Safe-Tk.
- *Tclets* do not have access to standard input and standard output.

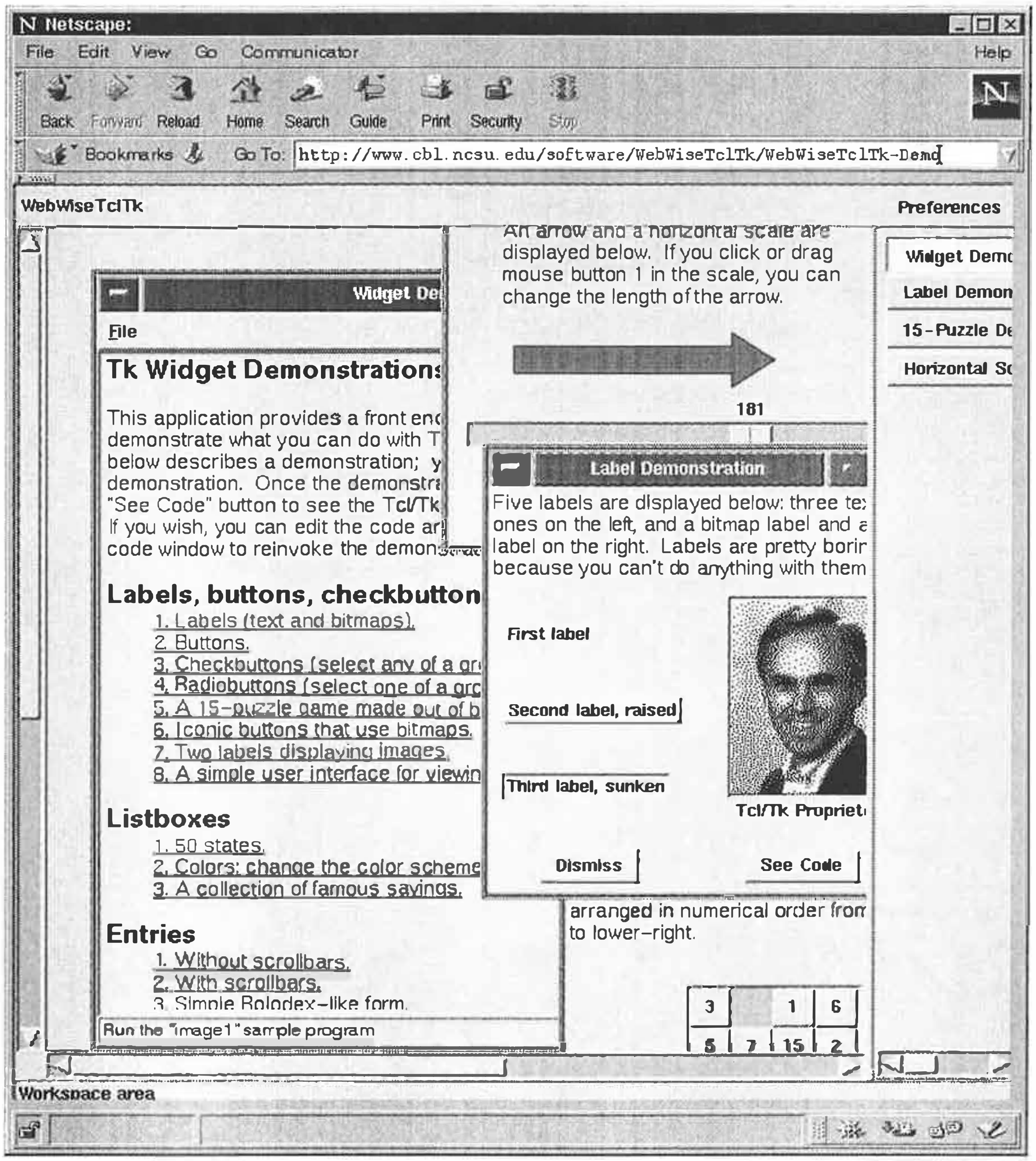
The Tcl-plugin supports multiple security policies so that the *tclets* can perform any of the functionality described above. However, this requires every client host to devise and customize their security policies for every application before accessing these as *tclets*.

It is desirable that the Tcl applications be easily translated into *tclets* and made readily available on the World Wide Web:

- without requiring any major changes in the application code, and
- without requiring any sophisticated security policy to run the *tclet*.

We have developed the WebWiseTclTk toolkit as an enhancement to the Tcl-plugin that makes use of the *home policy* only. The *home policy* is, by default, enabled in the Tcl-plugin and hence applications using WebWiseTclTk do not require the host clients to modify their existing security policies.

We decided to use the Tk widget demonstrations, distributed with the core Tcl/Tk, as a test-bench for testing the WebWiseTclTk toolkit. We chose to translate these demos for the World Wide Web because they cover most of the commands of the core Tcl/Tk that are otherwise unavailable in Safe-Tcl/Tk. Figure 2 shows the result of posting these demos on the Web and executing them on a host client as a *tclet* using the Netscape browser. We invite users to try out this demo and send us comments on its features and performance.



Tk Widget Demonstration

This application provides a front en
demonstrate what you can do with T
below describes a demonstration; y
demonstration. Once the demonstra
"See Code" button to see the Tcl/Tk
If you wish, you can edit the code ar
code window to reinvoke the demon

Labels, buttons, checkbutton

1. Labels (text and bitmaps).
2. Buttons.
3. Checkbuttons (select any of a gr
4. Radiobuttons (select one of a grc
5. A 15-puzzle game made out of b
6. Iconic buttons that use bitmaps.
7. Two labels displaying images.
8. A simple user interface for viewin

Listboxes

1. 50 states.
2. Colors: change the color scheme
3. A collection of famous sayings.

Entries

1. Without scrollbars.
2. With scrollbars.
3. Simple Rolodex-like form.

Run the "image1" sample program

An arrow and a horizontal scale are displayed below. If you click or drag mouse button 1 in the scale, you can change the length of the arrow.

Five labels are displayed below: three te ones on the left, and a bitmap label and a label on the right. Labels are pretty borir because you can't do anything with them

arranged in numerical order from to lower-right.

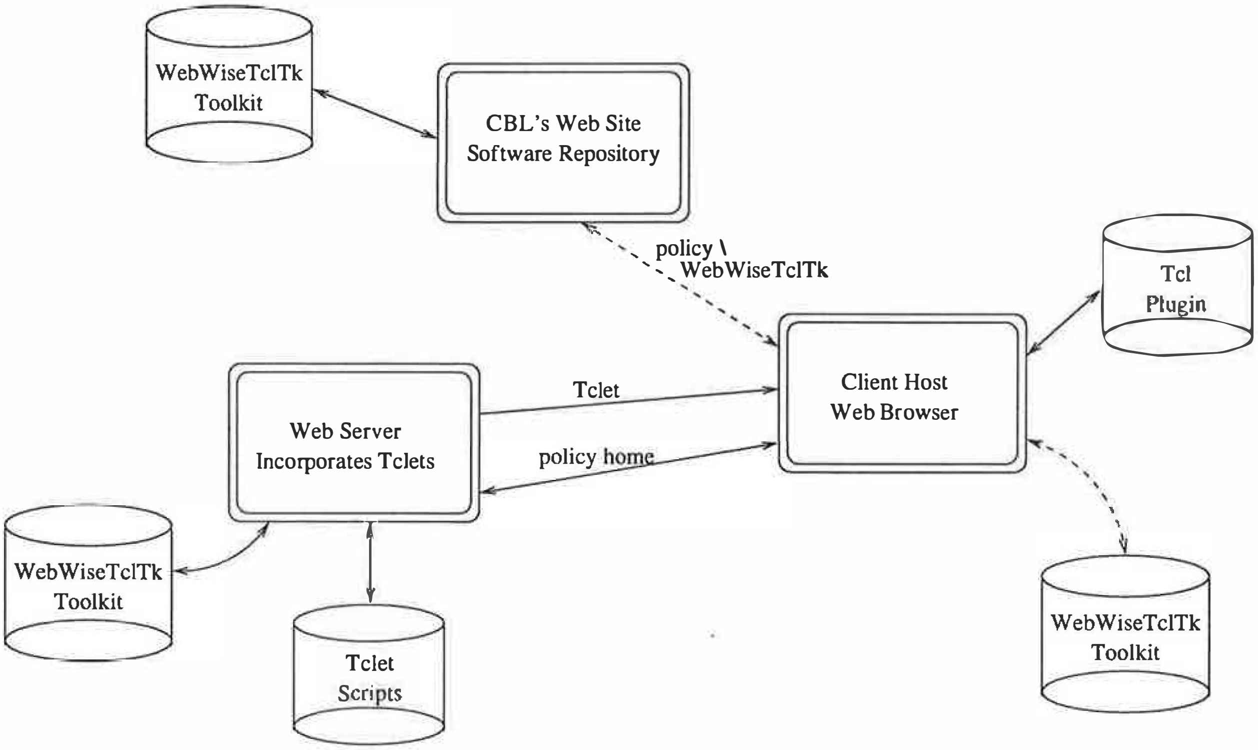
Fig. 2. Tcl/Tk widget demos on the Web.

3 Architecture

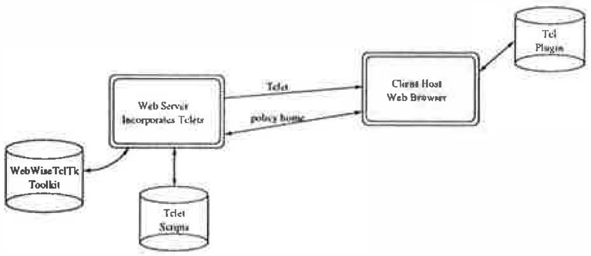
The toolkit `WebWiseTclTk` consists of two parts: (1) `WebWiseTcl` which is an enhancement for Safe-Tcl and is useful for applications that do not require display, and (2) `WebWiseTk` which is an enhancement for Safe-Tk for applications requiring display. The toolkit itself consists of several smaller scripts and uses the modified auto_load mechanism designed for `WebWiseTclTk`.

Figure 3(a) shows the general architecture that implements the auto_load mechanism. Special cases of the generalized architecture are shown in Figures3(b), (c) and (d) and described below:

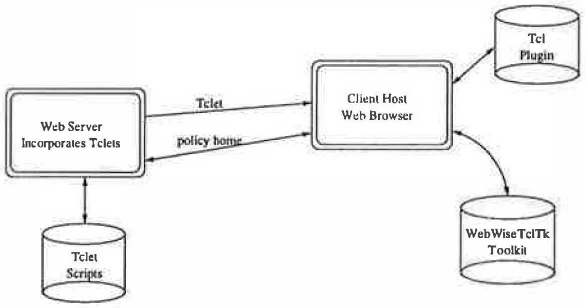
1. Typical client host, downloading a *tclet* from a Web server, has only the Tcl-plugin installed for its Web-browser. The server site provides not only the *tclet scripts* but also the `WebWiseTclTk` toolkit as shown in Figure 3(b). The client host downloads the main script for the *tclet* which requests to use



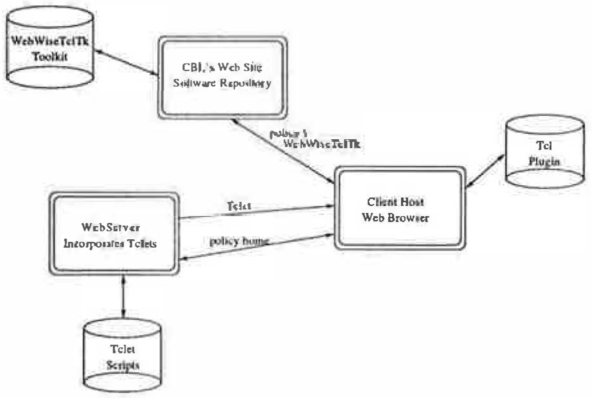
(a) Generalized architecture.



(b) `WebWiseTclTk` toolkit on the Web server.



(c) `WebWiseTclTk` toolkit on the client host.



(d) `WebWiseTclTk` toolkit on the CBL site.

Typical scenario of a *tclet* execution:
1. Client host visits the Web server.
2. Main *tclet* script is downloaded to client host.
3. `WebWiseTclTk` toolkit is loaded and initiallized.
4. Main *tclet* script executes.
5. Individual *tclet* scripts are downloaded as and when required.

Fig. 3. Architecture for `WebWiseTclTk` tookit.

the *home policy.* If the client host has not disabled the *home policy*, then the main script downloads the initialization script of the `WebWiseTclTk` toolkit from the server site. Once the initialization has completed, the auto_load mechanism is modified to dynamically download the remaining scripts of the application as and when needed during execution of the *tclet.*

2. In the second case, shown in Figure 3(c), the client host has locally installed the `WebWiseTclTk` toolkit. The main script of the downloaded *tclet* uses the locally available toolkit and visits the server site only to retrieve its other scripts. Thus, this results in faster execution of the *tclet* code.

3. In the third case, shown in Figure 3(d), the `WebWiseTclTk` toolkit is neither available on the server site nor is it installed on the client host. It is available at the software repository site at CBL. This requires the client host to install a special *WebWiseTclTk policy* that allows the *tclets* to download scripts not only from its server site but to also download the toolkit from the CBL site. This mechanism has the advantage of always using the latest version of the `WebWiseTclTk` toolkit.

The generalized architecture allows the main *tclet* script to dynamically use one of the above three mechanisms, based on the configuration of the client host.

We next describe the implementation details of the two packages `WebWiseTk` and `WebWiseTcl`.

4 WebWiseTk

Several Tk commands are hidden in Safe-Tk to prevent *denial of service* attacks against the host system. This, however, limits the scope of the Tcl-plugin to very simple applications consisting of a single window only.

We propose to overcome these limitations as follows:

- re-introduce several of the hidden commands in Safe-Tk,
- use existing commands that are already available in Safe-Tk, to define re-introduced commands,
- change the implications of a few commands, such as "`grab -local`" and "`grab -global`".

The following sub-sections describe the methodology used for implementation of the `WebWiseTk` toolkit.

Layout. Figure 4 shows the layout window of the `WebWiseTk` toolkit. It consists of two main widget frames:

- A canvas widget is used to display several toplevel windows that may be created during the execution of a *tclet*. If the toplevel window is larger than the visible canvas area, then scrollbars may be used to display its hidden area. The scrollregion of the canvas is initialized to 1000 × 1000 pixels, but may be resized using the preferences option on the top right corner.
- A text widget is used to display button icons for all windows that have been created, including those windows that have been iconified.

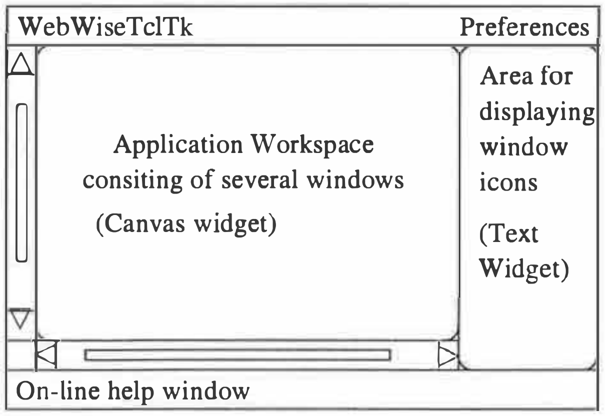


Fig. 4. Layout window of WebWiseTk toolkit.

Several other widgets are shown, such as the On-line help window, scrollbars for the canvas and text widgets, and preferences to configure the size of the canvas widget.

Toplevel. The ability to create a detached window, as provided by the command `toplevel`, is very useful for GUI applications of even moderate complexity. We define a procedure called `toplevel` which makes use of the command `frame` to create a detached window and display it on the canvas widget. For every toplevel window, a set of several frames is created, as shown in Figure 5.

This gives the look and feel of a real window that would have, otherwise, been created by the window manager of the local host system. The frames are laid out using the `grid` geometry manager. Each frame serves a special purpose:

- The frames on the border have a default color and an active color which is highlighted whenever the mouse cursor moves inside a window. This helps the user to identify the window that is currently active. These frames are also useful for changing the size of the window.
- The next set of frames, just below the resizing frame on the top, provide several functions related to the window, such as `kill`, `iconify`, `maximize/restore size` or display the title of the window.
- A main frame is created in the center corresponding to each toplevel window. All subsequent child windows of the toplevel are packed into this frame.

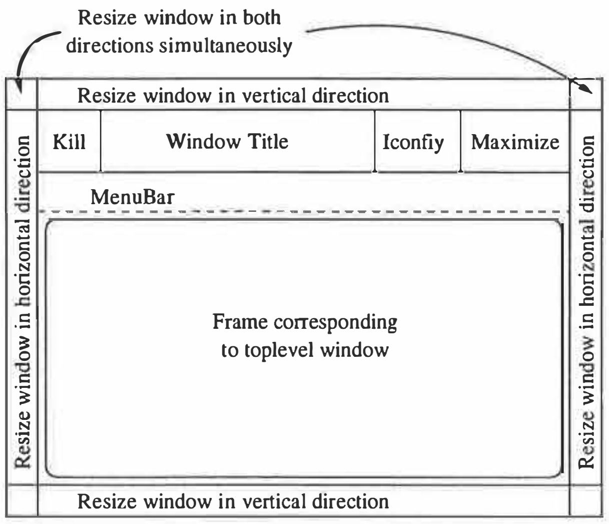

Resize window in both directions simultaneously
Resize window in vertical direction
Kill
Window Title
Iconfiy
Maximize
MenuBar
Frame corresponding to toplevel window
Resize window in horizontal direction
Resize window in horizontal direction
Resize window in vertical direction

Fig. 5. Implementation of a toplevel window using frames.

• If the toplevel window has a `Menu Bar` associated with it, then the menu items are packed into a frame just above the main frame.

Having created these sets of frames, they are then packed onto the canvas widget in the application workspace area in Figure 4 by creating a canvas item of type *window*. This results in a restriction that the pathname of the window must either be a child of the canvas widget or a child of some ancestor of the canvas widget. Hence, the window names of every toplevel command is translated to a new window name that is a child of the canvas widget. For example, a new toplevel window called "`.w`" is translated to a new window name called "`.c.1.w`", where "`.c`" is the name of the canvas widget and "`.c.1`" is the name of a unique frame created for encapsulation of the new toplevel window. In addition, we create several bindings to manage and interact with the command `wm`, described next. A `<Destroy>` binding is also associated with every toplevel frame so that the entire set of frames is destroyed whenever the toplevel window is destroyed.

Wm. The window manager command `wm` needs to be defined as a procedure which manages the various attributes of the window created using the procedure `toplevel` described earlier. It can be used to change the title of the window, to iconify/de-iconify the window, or to return the state of the window.

Grab. An indefinite global `grab` performed by a *tclet* will result in a *denial of service* attack since all the input from the terminal would be re-directed to the *tclet* forever. But, if we re-define the implication of a global `grab` such that it affects only the windows created by the *tclet*, then it can be considered to be safe.

Thus, the command "`grab -local $win`", as defined here, results in grabbing of a single window within the *tclet* code and the command "`grab -global $win`" results in a grab across all the windows within the *tclet*. This effect of grab can be implemented by associating a new class of bind called `WebWiseTclTk` with every window in the *tclet*, as follows:

```
bindtags $w [linsert [bindtags $w] 0 WebWiseTclTk]
```

Initially, the class `WebWiseTclTk` has no bind scripts associated with any of the events. Whenever a `grab` is performed on a window, a bind script is created for each event sequence that redirects the event to the grabbed window. The `event generate` command is used to process the event in the grabbed window. Figure 6 shows a script that achieves a global grab for a specific window. A grab is released by re-initializing the bind scripts for the class `WebWiseTclTk` to null. Variables are used to store the state of the `grab` command and return appropriate values for queries such as "`grab current`" and "`grab status`".

```
foreach seq $AllEventSequences {
  bind WebWiseTclTk $seq "
   # Redirect events to grab window
   if {!\[string match $win* %W] &&
          \[winfo exists $win]} {
     event generate $win $seq
     break
   }; # End of if stmt
  "
}; # End of foreach loop
```

Fig. 6. Script to achieve a global grab on a window.

Menus. Menu widgets are as important as any toplevel widgets in any GUI applications, since they allow the user to invoke a list of one-line entries as and when required. The structural layout of the `menu` widgets created using `frame` and other Tk commands is shown in Figure 7. The command `menu` creates a toplevel frame and different types of widgets are added inside this frame: `button` widgets for `command` entries, `checkbutton` widgets for `check button` entries, `radiobutton` widgets for `radio button` entries and `menubutton` widgets for `cascaded menu` entries. Separator entries are created using `frame` widgets as shown in Figure 7.

This structure is hidden from the display until the user clicks on the menu button at the top. The implementation of the command `grab`, as described earlier, is important and allows us to post the menu widget frame whenever the user clicks on the menu button. As the user moves the cursor over differ-

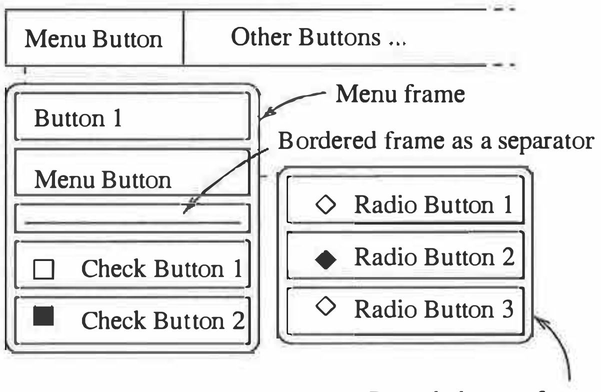
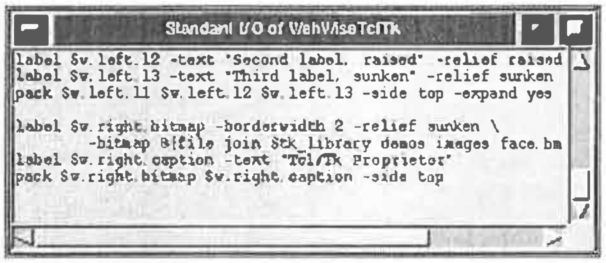


Fig. 7. Implementation of the command `menu`.

ent widgets in the menu frame, each widget is highlighted and the associated command invoked if necessary. Clicking on the cascaded entry results in the posting of another menu frame with its associated entries.
If the menu widget is of the type pulldown menu in a Menubar, then the menu entries are packed into the Menubar frame that was created in the `toplevel` procedure (Figure 5).



```
label $v.left.12 -text "Second label, raised" -relief raised
label $v.left.13 -text "Third label, sunken" -relief sunken
pack $v.left.11 $v.left.12 $v.left.13 -side top -expand yes

label $v.right.bitmap -borderwidth 2 -relief sunken \
        -bitmap @[file join $tk_library demos images face.bm
label $v.right.caption -text "Tcl/Tk Proprietor"
pack $v.right.bitmap $v.right.caption -side top
```

Fig. 8. Standard I/O of `WebWiseTclTk` toolkit.

Standard I/O and audio. We have created a special window for standard I/O in `WebWiseTclTk`. Any communication to the standard I/O channel by commands such as `puts` and `gets` is redirected to the special window, as shown in Figure 8. Therefore, it is possible for the *tclet* and a user to interact through the commands `puts` and `gets`.
Audio commands, such as `bell`, are still potentially dangerous, with the risk of producing a continuous tone. Therefore, we defined a procedure `bell` which produces a visual effect by momentarily changing the background color of the canvas widget.
Safe commands. Earlier, we noticed that whenever a toplevel window, say "`.w`", is created, the window name is mapped to a new window name "`.c.1.w`", corresponding to the main frame in the set of toplevel frames. Therefore, existing safe commands such as `button` with window names "`.w.b`" will fail, unless their window names are also translated to a new name "`.c.1.w.b`", which is in the hierarchy of the toplevel main frame's children.
We rename the existing safe commands by moving them into a namespace for `WebWiseTk`, and define new procedures for them. Figure 9 shows a sample code for re-defining the command `button`. The newly defined procedure does the following:
1. maps all the window names, in the arguments passed to the procedure, to the corresponding hierarchy in the toplevel frame.
2. evaluates the original command `button` with mapped arguments. This creates a new command "`$new_w`" for the translated window pathname.
3. defines a new procedure for the original window pathname "`$w`" that would have been created otherwise. This procedure, in turn, invokes "`$new_w`" whenever it is called.
4. translates the window names in the returned values back to original window names. This is important because the returned values may be directly passed to other code for evaluation. Example: "`pack [button .b]`".
5. returns the translated value "`$new_ret`".

```
# Move original command to WebWiseTk namespace
rename button ::WebWiseTk::button-Org
# Define a new command in the global namespace
proc ::button {w args} {
# Map window names
 set new_w [mapArgWindowNames $w]
 set new_args [mapArgWindowNames $args]
 set ret [uplevel 1 ::WebWiseTk::button-Org \
             $new_w $new_args]
# Create a proc called $w
 proc ::$w {args} {
   # Script to invoke command $new_w ...
 }
# Map back returned window names
 set new_ret [mapRetWindowNames $ret]
 return $new_ret
}
```

Fig. 9. New definition for command `button`.

The command `bind` also has to be re-defined. This is because the value of "`%W`" in the bind script gets the real window name ("`.c.1.b`") instead of the window name ("`.b`") supplied by the *tclet*. Thus all window names referred by "`%W`" in the bind script are mapped back appropriately, before invoking the original bind script.
Similarly, the command `winfo` is also redefined, so that its queries, such as "`winfo width`", "`winfo children`", etc., are correctly handled.
Unsafe commands. Few commands, such as `send`, `tk_getOpenFile`, `tk_getSaveFile`, etc., do not pose

the *denial of service* attacks, but are still unsafe and very dangerous to the client host system since they present other forms of security attacks. These commands are therefore not available in `WebWiseTk`. However, it is always possible to use an appropriate security policy, other than the *home policy*, to enable these commands.

Unsafe options. A few options for safe commands are considered unsafe and hence not available in Safe-Tk. These include `"-bitmap @filename"`, `"-file filename"` and `"-maskfile filename"`, among others. It is possible to allow these options on the following conditions:

1. the host system supports the use of the *home policy*, and
2. the specified file exists on the server site of the *tclet* code.

In such a case, the data for the specified filename is downloaded from the server site using the command `"::browser::getURL filename"`. Then `"-file filename"` or `"-maskfile filename"` option is replaced by `"-data $downloadedData"` or `"-maskdata $downloadedData"`. On the other hand, for the option `"-bitmap @filename"`, a bitmap image is first created using the command `image`. Here the replaced option is `"-image [image create bitmap -data $downloadedData]"`. The command `"image create image -file filename"` is also replaced with `"-data $downloadedData"` option, after we download the data for the specified filename from the server site. However, the option `"-data $downloadedData"` expects the image data to be in *base64* format. Images in other formats are therefore tranlated to base64 format using the *tcl-only* encoding procedures available in the Data Handling Package [15]. The enocoding process is slow and hence for images of considerable size, one should save the original images in *base64* format, instead of encoding them on the fly during execution of the *tclet*.

5 WebWiseTcl

To maintain security, it is important that the unsafe Tcl commands be hidden or restricted in Safe-Tcl. Several different security policies offered by Tcl-plugin 2.0 are convenient and allow the application programmer to design *tclet* codes accordingly. We intend to make use of the *home policy* to enhance the functionality of the Safe-Tcl for `WebWiseTcl`.

Script libraries and packages provide an excellent mechanism to structure an application code into several smaller scripts, and then dynamically load each script as needed. We modify the restricted commands in Safe-Tcl such that it supports the packaging facility to automatically load scripts from the server site of the *tclet* code. We only need to append the location of the server site, given by `"getattr originHomeDirURL"`, to the `auto_path` variable for the `auto_load` procedure to work correctly with the modified commands described next.

Source. The filename argument for the `source` command is parsed for a URL. If the filename is a URL, then it is downloaded using the command `"::browser::getURL filename"` and its contents are evaluated. Otherwise, the original `source` command is invoked, as shown in Figure 10.

```
# Move original command to WebWiseTcl namespace
rename source ::WebWiseTcl::source-Org
# Define a new command in the global namespace
proc ::source filename {
 if {[string match http:* $filename]} {
  # Evalute script downloaded from a URL
  uplevel 1 [::browser::getURL $filename]
 } else {
  # Invoke original source command
  uplevel 1 ::WebWiseTcl::source-Org $filename
 }
}
```

Fig. 10. New definition for command `source`.

Open and close. When a filename specified for `open` is a URL, the specified URL is downloaded and saved on the temporary disk space of the host system assigned by the *home policy*. Then, this file on the local disk is opened and its channel identifier returned. Correspondingly, when a `close` command is invoked for a URL, the file on the local disk is not only closed, but also deleted. These functions are useful for opening a file/URL in read-only mode.

File. We have re-defined the command `file` so that its options `dirname`, `join`, and `split` return correct results even when the specified filename is a URL.

Pwd, cd and glob. These commands are not available in Safe-Tcl. We therefore assign the URL of the server site, given by `"getattr originHomeDirURL"`, to be the default working directory returned by the command `pwd`. This value is stored in a variable defined in `WebWiseTcl` namespace. The invocation of the command `cd` then results in change of value of the current working directory stored in the variable. The command `glob` returns a list of all matching URLs found under the URL given by the current working directory.

6 Users Guide

We define *users* as those who intend to download and view `WebWiseTclTk` toolkit-based Tcl-plugin appli-

cations within their Web browsers.
Users can very easily and quickly familiarize themselves with the WebWiseTclTk environment. Figure 2 shows one such typical view of the environment within a Netscape browser. The layout of the environment is shown in Figure 4. It has two widget areas - the one on the left contains windows created by the *tclet*, and the one on the right displays a list of buttons corresponding to each iconified window. Both the widgets have auto scrollbars. At the bottom, a single line help message is displayed, based on the location of the mouse cursor. The size of the widget containing the *tclet* windows may be increased or reduced by the user under the Preferences option. A user may also resize the canvas and the text widget areas by merely dragging the border between the two with a mouse cursor.

Installation. It is not necessary for the users to install the WebWiseTclTk toolkit. The scripts in the toolkit are dynamically downloaded, as and when required, from the server site of the running application/ *tclet.* However, for faster access, users do have an option of installing the WebWiseTclTk toolkit in their Tcl-plugin directory. In this case, the installation procedure consists of the following:

1. Download the latest version of the WebWiseTclTk toolkit from:
`http://www.cbl.ncsu.edu/software/#WebWiseTclTk.`
2. Change to the installation directory of the Tcl-plugin on your local file system.
```
  # For local installation,
    csh% cd ~/.netscape/tclplug/2.0
  # Or, for site installation,
    csh% cd /usr/local/lib/netscape/tclplug/2.0
```
3. Gunzip and untar the toolkit.
```
 csh% gzip -dc WebWiseTclTk-x.y.tar.gz | tar xf -
```
4. Verify the installation by visiting the test site under `http://www.cbl.ncsu.edu/software/#WebWiseTclTk.`

The toolkit consists of *tcl-only* scripts and hence does not require any compilation step.

7 Programmers Guide

We define *programmers* as those who: (1) intend to write Tcl-plugin applications based on the WebWiseTclTk toolkit, or (2) wish to translate their existing Tcl applications into *tclets* for execution over the Web.
Programmers, who intend to use the WebWiseTclTk toolkit for their *tclets*, should follow the guidelines listed below:

1. Download the latest version of the WebWiseTclTk toolkit from:
`http://www.cbl.ncsu.edu/software/#WebWiseTclTk.`
2. Change to a directory on your system that is accessible on your Web server.
```
    csh% cd /home/user/public_html/tclets
```
For example, let the URL corresponding to this directory be `http://www.your.web.site/~user/tclets.`
3. Gunzip and untar the toolkit
```
    csh% gzip -dc WebWiseTclTk-x.y.tar.gz | tar xf -
```
4. Verify the installation by clicking visiting the examples distributed with the toolkit under
`http://www.your.web.site/~user/tclets/`
`WebWiseTclTk-x.y/examples`

Figure 11 shows an example to encapsulate the Tk widget demos and execute them on the Web. The TkWidgetDemos.tcl script, the demos directory and the WebWiseTclTk toolkit directory all exist in the same directory location on the Web as shown below:

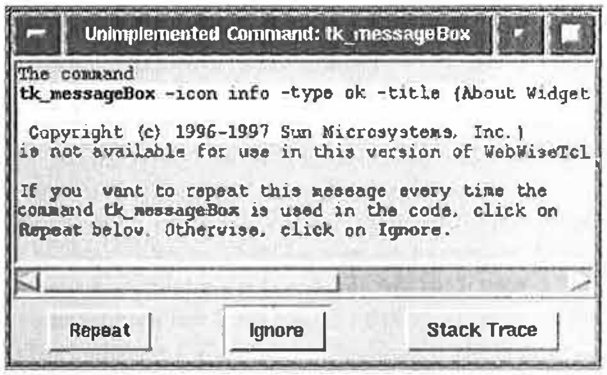
```
/home/user/public_html/tclets/WebWiseTclTk-x.y/examples
          \
           |_ TkWidgetDemos.html
           |_ TkWidgetDemos.tcl
           |_ WebWiseTclTk
           |              \
           |               |_ DownloadToolkit.tcl
           |               |_ toplevel.tcl
           |                     ...
           |_ demos
                  \
                   |_ widget
                   |_ arrow.tcl
                   |_ button.tcl
                        ...
```

When a user downloads the TkWidgetDemos.tcl script, the script first tries to load the WebWiseTclTk toolkit from the user's host system. If it succeeds, then the *home policy* is requested since the Tk widget demos consist of several different scripts. On the other hand, if the WebWiseTclTk toolkit cannot be loaded from the user's host system, then it is downloaded from the *tclets*'s server site.
The variable $tk_library is set to point to the *tclets*'s server site so that it knows from where to auto_load the demo script. Finally, the widget script is sourced to execute the demos.
If the *tclet* does not require the use of display, possible by setting "tk=0" in the html embed statement, then it is also possible to load only the WebWiseTcl toolkit.

Debugging. When writing new *tclets*, programmers can avoid using *Tcl-commands* which are either *not available* or *not yet implemented* in WebWiseTclTk toolkit. However, when converting existing applications, it is very difficult to isolate and remove these commands in the code. Therefore, we provide a mechanism whereby a dialog box is popped up whenever any *unavailable* or *unimplemented* command is used in the code the first time, as shown in Figure 12.
The programmer, who is testing the *tclet* as a user,

```
# TkWidgetDemos.tcl --
#

set WebWisePKG WebWiseTk  ; # Necessary, if the tclet requires display
#set WebWisePKG WebWiseTcl ; # Use this if display is NOT required
#
# Get the URL for tclet's server site
if {[catch {set originHomeDirURL [string trimright [getattr originHomeDirURL] /]}]} {
 # Software is invoked from the local file system (NOT in a Web browser)
 set originHomeDirURL /home/your_path/tclets
 set auto_path [linsert $auto_path 0 $originHomeDirURL/WebWiseTclTk]
 package require $WebWisePKG
} else {
 # Software is invoked from a Web browser
 # Check whether the toolkit is available locally on the user's host system.
 if {[catch {package require $WebWisePKG}]} {
  # Not available - so download it from the tclet's server site.
  policy home      ; # Need this policy to fetch URL from the server site.
  eval [::browser::getURL $originHomeDirURL/WebWiseTclTk/DownloadToolkit.tcl]
  # Now setup the package auto_load
  package require $WebWisePKG
 } else {
  # Use the home policy, if the tclet code consists of several scripts
  policy home
 }
}
# Initiallize the WebWiseTk layout
WebWiseTk .webdesk

# Set tk_library, so that it can access other files in the demos directory.
set tk_library $originHomeDirURL
# Now invoke the Tk widget demos
source $originHomeDirURL/demos/widget

# Alternatively, if the tclet code is small, it can follow here.
# your tcl script...
```

Fig. 11. Main script for encapsulation of the Tk widget demos shown in Figure 2.



```
The command
tk_messageBox -icon info -type ok -title {About Widget

 Copyright (c) 1996-1997 Sun Microsystems, Inc.}
is not available for use in this version of WebWiseTcl

If you  want to repeat this message every time the
command tk_messageBox is used in the code, click on
Repeat below. Otherwise, click on Ignore.
```

Fig. 12. Dialog box for unimplemented commands.

has a choice to either ignore the generation of this dialog box, the next time the same command is used, or to repeat it. A "Stack Trace" button is also available to locate the generation of the unimplemented command in the code.

Extensibility and reconfigurability. The `WebWiseTclTk` toolkit consists of several smaller scripts, specifically one file for each command that is either newly defined or re-defined. Therefore, programmers can (1) define and add their own commands which may be unavailable, such as `send`, in a separate file, or (2) re-define the existing commands by modifying the corresponding file to implement their own version of the command.

For example, the commands `toplevel` and `menu` may be re-implemented with a different layout to give a native look and feel on different platforms.

8 Experiences and Future Scope

The *Tcl-only* implementation, mapping window names onto a canvas object, real-time conversion of gif images to base64 format, etc results in degradation of performance. We have tried to minimize these effects by modularizing the code in such a fashion that related tcl procedures are clustered into single script files. Thus, procedures required for implementation of the `menu` command are loaded only if the command is used by the *tclet* code. Another area of optimization is possible by improving the efficiency of the procedure to map window names back and forth onto canvas objects, since this is one of the

most frequently called procedure.
Installing a local copy of the toolkit with the Tcl-plugin on the client host will improve the performance when the distance between the client host and the web-server is large. We also need to improve the reliability of the toolkit by adding sufficient hooks to handle cases when 'getURL' is likely to timeout or fail under high network traffic conditions.

9 Software Availability and Status

The `WebWiseTclTk` toolkit described is available at `http://www.cbl.ncsu.edu/software/#WebWiseTclTk`. The current version of the toolkit is *beta 1.02*.
We have successfully tested this version of the toolkit on a Sun Sparc workstation with *Solaris 2.5.1* using Netscape 4.0 and 3.0.
On a *Windows 95/NT* machine, we had to install the toolkit locally before we could access *tclets* based on `WebWiseTclTk`. Also, we had to use a special policy that allows downloading scripts from the server site using the command `::http::geturl` from the http package. This is because, (1) the blocking version of the command `::browser::getURL` is not supported in Netscape 4.0, and (2) the command `::browser::getURL` is not available under Internet Explorer 4.0 for the Tcl-plugin. The implementation of menu widgets in `WebWiseTclTk` is specific to Unix and hence do not function properly on a *Windows 95/NT* machine.
We have also tested the toolkit successfully on a *Mac* running under *MacOS 8.0*. Again, the current implementation of menu widgets do not work correctly on a *Mac*.
Some of the features of the `WebWiseTclTk` toolkit described in this paper are not yet implemented. For example, in menu widgets, advanced features that are not implemented include: the `accelerator` option for any of its entry is ignored, creation of clones of menu frames using the tear-off entry is not possible, etc. For details of such items and current updates, please consult
`http://www.cbl.ncsu.edu/software/#WebWiseTclTk`.

10 Conclusions

We have demonstrated the capabilities of the `WebWiseTk` toolkit by readily posting an existing *Tcl application*, namely the Tk Widget Demos, as an executable *Tclet* on the Web. Since these demos cover most of the commands available in the main Tcl/Tk interpreter, they signify the potential usefulness of the toolkit.
Introduction of the `WebWiseTcl` toolkit, which uses the *home policy*, enables *programmers* to structure their *tclets* into several smaller scripts. Such scripts are easier to manage and dynamically loaded during the execution of the *tclet*.
While most of the commands related to *denial of service* attacks may be eventually restored in the Tcl-Plugin, `WebWiseTclTk` toolkit still offers the ability to confine the *tclet* windows to a single display within the Web browser.
Our first major application of `WebWiseTk` has been the introduction of Web-based user-configurable and executable workflows that support an environment functionally similar to one in `REUBEN` [11, 12, 13]. First demos of this capability has been shown in the University Booth during the 1998 Design Automation Conference [16]. On-line demos are accessible from `http://www.cbl.ncsu.edu/demos`.

References

[1] The Tcl/Tk Home Page. Published under URL `http://sunscript.sun.com/`, 1997.
[2] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
[3] B. B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 1997.
[4] J. K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. Published under URL `http://scriptics.com/people/john.ousterhout/-scripting.html`, March 1998.
[5] Scriptics Corporation. Published under URL `http://www.scriptics.com/`, 1998.
[6] The Tcl/Tk Consortium. Published under URL `http://www.tclconsortium.org/`, 1998.
[7] The Tcl Plugin Home Page. Published under URL `http://sunscript.sun.com/plugin`, 1997.
[8] J. Y. Levy. A Tcl/Tk Netscape Plugin. Published under URL `http://sunscript.sun.com/plugin/-paper.html`, May 1996.
[9] J. K. Ousterhout, J. Y. Levy, and B. B. Welch. The Safe-Tcl Security Model. Published under URL `http://scriptics.com/people/john.ousterhout/-safeTcl.ps`, March 1997. Draft.
[10] D. Libes. *Exploring Expect*. O'Reilly and Associates, 1995.
[11] H. Lavana, A. Khetawat, F. Brglez, and K. Kozminski. Executable Workflows: A Paradigm for Collaborative Design on the Internet. In *Proceedings of the 34th Design Automation Conference*, pages 553–558, June 1997. Also available at `http://www.cbl.ncsu.edu/publications/-#1997-DAC-Lavana`.
[12] Amit Khetawat. Collaborative Computing on the Internet. Master's thesis, Electrical and Computer Engineering, North Carolina State University, Raleigh, N.C., May 1997. Also available at `http://www.cbl.ncsu.edu/-publications/#1997-Thesis-MS-Khetawat`.
[13] H. Lavana, A. Khetawat, and F. Brglez. Internet-based Workflows: A Paradigm for Dynamically Reconfigurable Desktop Environments. In *ACM Proceedings of the International Conference on Supporting Group Work*, Nov 1997. Also available at `http://www.cbl.ncsu.edu/-publications/#1997-GROUP-Lavana`.
[14] I. K. Lam and B. Smith. Jacl: A Tcl Implementation in Java. In *Proceedings of the Fifth Annual Tcl/Tk Workshop*, July 1997.
[15] Document Handling Package. Published under URL `http://tcltk.anu.edu.au/DHP/`, 1997.
[16] Design Automation Conference. Published under URL `http://www.dac.com/`, 1998.

Internet-based Desktops in Tcl/Tk: Collaborative and Recordable

Amit Khetawat Hemang Lavana Franc Brglez
CBL (Collaborative Benchmarking Lab), Dept. of Computer Science, Box 7550
NC State University, Raleigh, NC 27695, USA
http://www.cbl.ncsu.edu/

Abstract

This paper addresses issues that arise when a peer group, distributed across several time zones, uses the Internet to configure and execute distributed desktop-based applications and tasks. The paper provides solutions and Tcl/Tk implementations to support (1) peer-to-peer communication/control of distributed software and computing resources over the Internet; (2) recording and playback of interactive execution of Tcl/Tk applications and collaborative sessions.

The summary of 540 Internet-based experiments, each relying on RecordTaker *and* PlaybackMaker *to record, playback, and execute* ReubenDesktop *configurations from local, cross-state, and cross-country servers, demonstrates the effectiveness of the proposed concepts and implementation.*

Keywords: collaborative desktops, recording, playback, workflows, Internet.

1 Introduction

The Internet and the on-going evolution of the world-wide web is expected to evolve into a network without technologic, geographic or time barriers – a network over which partners, customers and employees can collaborate at any time, from anywhere, with anyone. Issues of collaboration arise when a peer group, distributed in time and space, uses the Internet to iteratively configure and execute distributed desktop-based applications and tasks. Existing systems, based on Tcl/Tk [1, 2, 3], that are available today include the the Group-kit [4, 5] for collaboration and the TkReplay [6] for recording.

Requirements that motivated this research resulted in collaborative and recording Tcl/Tk architectures and implementations that are different from the ones introduced earlier in [4, 5, 6]. Specifically, the Internet-based environment *ReubenDesktop*, prototyped in Tcl/Tk and Expect [7], leverages a novel multi-user collaborative desktop architecture that also supports multi-cast visualization of workflows, encapsulating distributed data, tools and communication protocols. A hierarchy of such workflows can be re-configured and re-executed by a team of collaborating users, since the *generic architecture* supports (1) effective channels of communication among peers, and (2) distributed control of applications and tasks. Data and tools encapsulated in such workflows reflect the needs of a specific peer group. The workflows decribed in this paper are representative of ones that may be used in the design of experiments or as an integral a part of a distributed microsystems design effort [8, 9, 10, 11, 12].

This research was supported by contracts from the Semiconductor Research Corporation (94–DJ–553), SEMATECH (94–DJ–800), and DARPA/ARO (P–3316–EL/DAAH04–94–G–2080 and DAAG55–97–1–0345).

In addition, we introduce a novel architecture that allows us to *record and playback* any Tcl/Tk application, including peer interactions during the *distributed and collaborative* sessions of *ReubenDesktop*.

The paper is organized into the following sections: (2) background and motivation, introducing a simple example of collaborative and recordable desktop environment; (3) collaborative Internet-based desktop environment, user view of *FlowSynchronizer* and its implementation; (4) session recording and playback environment, user view of *RecordTaker* and *PlaybackMaker* and their implementation; (5) a summary of 540 Internet-based experiments; (6) software availability and status; (7) conclusions and future work.

2 Background and Motivation

The *ReubenDesktop*, described in this paper as recordable and executable upon playback, satisfies the following properties as a collaborative desktop environment [9]:

P1: desktop is shared and multi-cast, so that each participant can observe desktop actions of others;

P2: desktop supports a shared and segmented 'talk window', so each participant can type messages to all others in his/her own window segment;

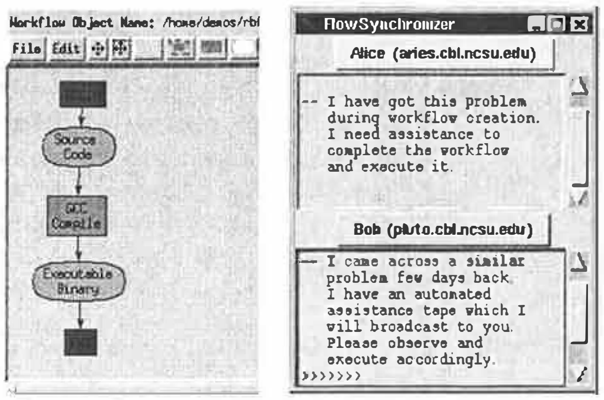
P3: the shared and segmented 'talk window' supports a token passing mechanism, so that at any time, only *a single user* controls the desktop, but can pass the token to any other user when requested.

An example of a *ReubenDesktop* satisfying properties **P1–P3** is shown in Figure 1(a). The instance of the particular desktop has been multi-cast by student Alice to her instructor Bob with a request for on-line assistance. In the case shown, the desktop consists of two windows: (1) a sample workflow that is not executing, hence the problem, and (2) a *FlowSynchronizer* window that allows Alice and Bob to 'talk' and describe the problem and a solution.

Here, instructor Bob could have requested and received permission from Alice to edit the workflow and thus show a solution. Instead, Bob remembers that earlier, he *recorded* a solution to a similar problem for another student. Subsequently, he decides to *playback* the pre-recorded solution, shown in Figure 1(b). By passing control to Alice (the respective *FlowSynchronizer* window is not shown), Alice can now study the solution by re-executing the *PlaybackMaker*.

It is clear that the paradigm described in this example applies to a number of situations, including design reviews, with high potential to reduce design errors or catch them early in the process, thereby significantly enhancing the productivity of the team effort.

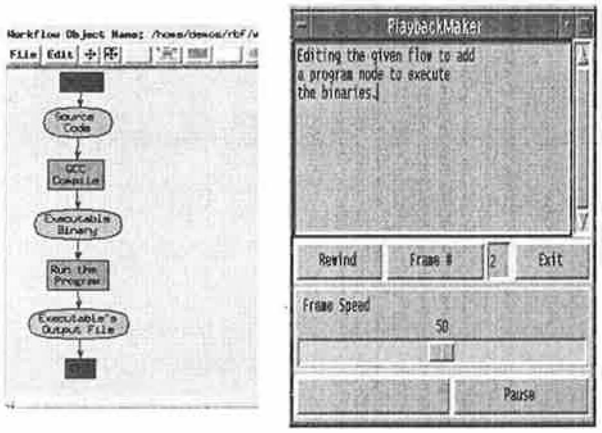
(a) Collaborative description of a problem



-- I have got this problem
during workflow creation.
I need assistance to
complete the workflow
and execute it.

-- I came across a similar
problem few days back.
I have an automated
assistance tape which I
will broadcast to you.
Please observe and
execute accordingly.
>>>>>>>

(b) Collaborative playback of a tutorial workflow



Editing the given flow to add
a program node to execute
the binaries.

Fig. 1. Example of a collaborative remote assistance.

3 Collaborative Internet-based Desktop Environment

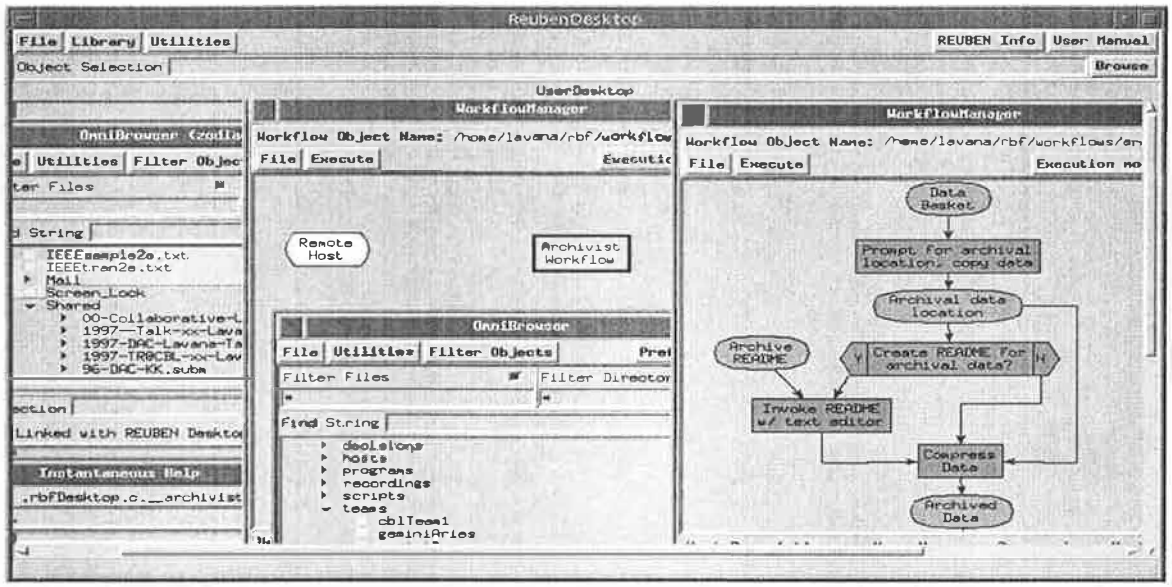
The Internet-based desktop environment as defined in [8] contains a number of application icons (program nodes) as well as a number of data icons (file nodes). In contrast to typical desktops of today, where data icons may be dragged and dropped onto application icons for execution, this environment allows

- *user-defined and reconfigurable execution sequences* by creating dependency edges between program nodes (application icons) and file nodes (data icons);
- *data-dependent execution sequences* by dynamic scheduling of path as well as loop executions;
- *host-transparency* as to the location of applications and data (both can reside on any host with a unique IP address).

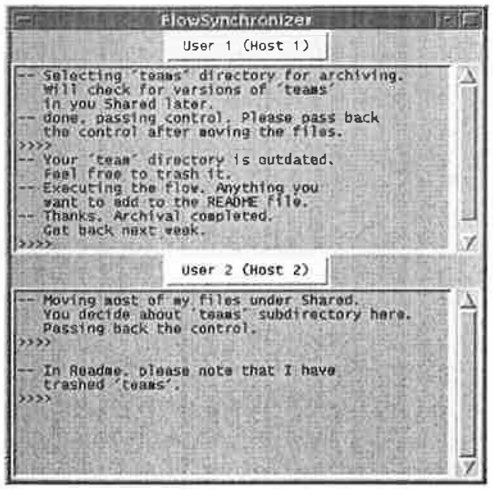
User View. We use a simple example of an *Archivist Workflow* to illustrate the concepts of a collaborative Internet-based desktop environment as defined in this paper. The *ReubenDesktop* in Figure 2(a) is a snapshot of a collaborative session in progress. Two archival specialists, User1 at Host1 and User2 at Host2, are completing the tasks of archiving distributed directories to a central location (Host1). The session was initiated by User1. Upon invocation of the *ReubenDesktop*, User1 gained access to the UserDesktop window within the ReubenDesktop and the *OmniBrowser*. Using *OmniBrowser*, she selected the *TeamDefinition* configuration file, and with a single click, she initiated the session with User2. At this point both receive an identical view of the *ReubenDesktop*, along with the *FlowSynchronizer* in Figure 2(b). User1 selects the *Archivist Workflow* in the OmniBrowser which is now broadcast as two nodes in the new workflow window within the desktop: 'Remote Host' and a hierarchical workflow node named *Archivist Workflow*. This node will execute whenever any of the files selected in the OmniBrowser window is 'dropped' onto it.

As noted in the *FlowSynchronizer*, User1 initiates the archival process, by selecting directory 'teams' and passes the control to User2. At this point, User2 clicks on the 'Remote Host' which invokes another *OmniBrowser*. He selects a set of files in the 'Shared' directory and passes back the control. User1, by

(a) Archivist Workflow within ReubenDesktop



(b) Archivist FlowSynchronizer



```
-- Selecting 'teams' directory for archiving.
   Will check for versions of 'teams'
   in you Shared later.
-- done. passing control. Please pass back
   the control after moving the files.
>>>>
-- Your 'team' directory is outdated.
   Feel free to trash it.
-- Executing the flow. Anything you
   want to add to the README file.
-- Thanks. Archival completed.
   Get back next week.
>>>>
```

```
-- Moving most of my files under Shared.
   You decide about 'teams' subdirectory here.
   Passing back the control.
>>>>

-- In Readme, please note that I have
   trashed 'teams'.
>>>>
```

(c) About FlowSynchronizer

The *FlowSynchronizer* can support n collaborating sites, providing controlled access to two window segments at each site: (a) a token button designating the *UserSite*, and (b) a message box which provides a real-time conferencing environment.

At any time, one and only one site is designated as a *Token-Holder*, by coloring its *UserSite* button in a color different from all other sites. At any time, each collaborator can transmit text in the message box to all other sites. However, only the *Token-Holder* has the capability to click on another *UserSite* button to pass the token, and hence the control of the entire environment, including any application and data displayed in the workflow window.

Fig. 2. Internet-based collaborative desktop environment.

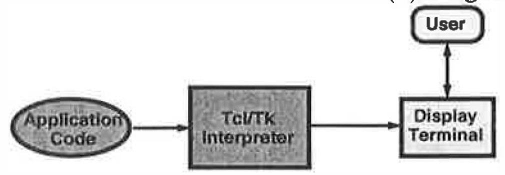
clicking on the *Archivist Workflow* node, 'drops' *all* files selected by *both* users onto this node, initiating the execution of the workflow – now shown fully expanded in the *ReubenDesktop*.

In this example, the Internet-based desktop environment has been rendered collaborative by the addition of a *FlowSynchronizer* that provides both the communication channels between collaborating participants and a control passing mechanism, supporting and maintaining the properties **P1**–**P3** summarized earlier. The functionality of the *FlowSynchronizer* is explained in Figure 2(c).

Collaborative Desktop Architecture. Two architectural extremes are possible to support collaborative activities:

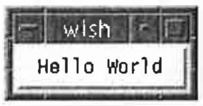
1. *Replicated software architecture* - exact copies or replicas of the application being shared must be installed and maintained on each host. The application on each host handles the user interaction locally and changes made to the application state are broadcast to all other replicas to maintain the consistency of the data and the user views. GroupKit [4, 5] is

(a) Single-cast Tcl/Tk architecture

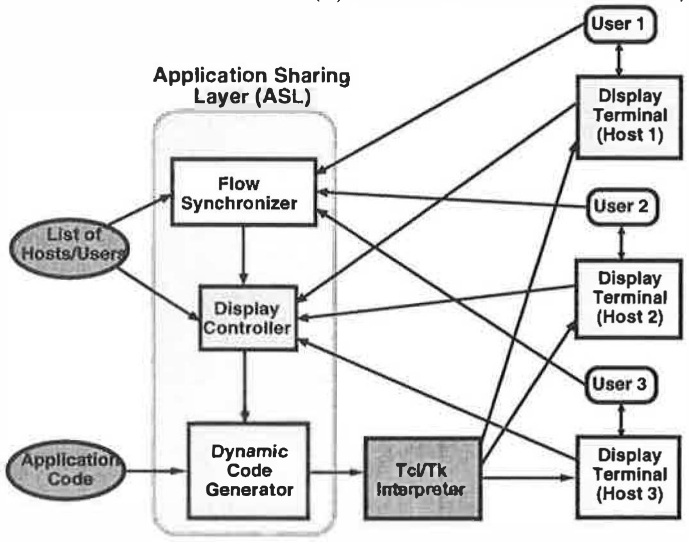


Original Application (Hello):

```
label .hello -text "Hello World"
pack .hello
```



(b) Multi-cast collaborative Tcl/Tk architecture



Multi-cast Application (M-Hello) generated by ASL:

```
# Initiallization script:
  toplevel .screen1 -screen screen@host1:0
  toplevel .screen2 -screen screen@host2:0
  toplevel .screen3 -screen screen@host3:0
# Dynamic code generation:
  label .screen1.hello -text "Hello World"
  label .screen2.hello -text "Hello World"
  label .screen3.hello -text "Hello World"
  pack .screen1.hello
  pack .screen2.hello
  pack .screen3.hello
```

Fig. 3. Multi-user *collaborative* desktop architecture.

an example of Tcl/Tk extension that follows this architecture and allows the development of Groupware applications such as drawing tools, editors and meeting tools. The major disadvantage of replicated architecture is the difficulty in keeping the data and user views consistent and in synchronization.

2. *Centralized software architecture* - a single host is responsible for setting up the collaborative session with the participating hosts which do not maintain any copies of the application being shared. Their local machines are nothing more than graphical terminals running X windows (on any UNIX, PC, or MAC workstation). The host starting the session handles all the incoming events in the form of user interactions and is responsible for sending and updating the shared applications views on all other participating hosts. The primary advantage of the centralized approach is its simplicity, since maintenance of the consistency of the application is related to a single host only.

We use *tcl-only* code to implement the centralized approach. In single-cast mode of operation shown in Figure 3(a), the application code is passed onto to the Tcl/Tk interpreter. The interpreter then makes appropriate function calls to display the *graphical user interface* (GUI) on the users display terminal. For multi-casting such a GUI to 'n' display screens, we modify the approach in Figure 3(a) to create a multi-cast collaborative mode of operation, by inserting an *Application Sharing Layer* (ASL) between the Tcl/Tk application code and the Tcl/Tk Interpreter, as shown in Figure 3(b). The ASL takes the original application code and a list of hosts as input. This layer consists of three components:

1. *FlowSynchronizer* - It provides the ability for real-time conferencing among the collaborating users.

2. *Display Controller* - It facilitates and manages ownership and exchanges of control over the application's GUI among the collaborating users.

3. *Dynamic Code Generator* (DCG) - This provides the ability to broadcast the application GUI to 'n' displays and also process events generated from the collaborators' displays.

Control Mechanisms. When a GUI of an application is broadcast to 'n' displays, it is necessary to coordinate user interaction with the application. If all the users try to interact with the application simultaneously, chaos and confusion is likely to follow soon. In most applications, however, it is essential that at any time, only one user has a control over the execution of the application. There are several mechanisms to avoid such contentions, which include: (1) single user interaction, with other users observing, (2) round-robin based transfer of control, allowing each user to interact for a limited time only, (3) request-based control, with users deciding as to who is to interact with the application at any instant, etc.

We have implemented *request-based control* mechanism where a single user is initialized with control over the entire application that is launched. The user who has the control and can interact with the application is said to hold the *token* and is called the *Token Holder*. At any instant, the Token Holder can relinquish the control over the application by passing the *token* to any other user. The Display Controller, shown in Figure 3(b), implements this scheme by: (1) accepting and responding to user interactions such as mouse movements, keyboard events, etc., from the Token Holder only, and (2) blocking all events that are generated by any other user.

FlowSynchronizer. Once all the users have a shared view of the application, they still need a mechanism to communicate among themselves for effective collaboration. If a user needs to request the control of the application from the current Token Holder, she has to communicate the request to the Token Holder. We provide an additional window on each users display, called the *FlowSynchronizer* shown in Figure 2(b), which allows the users to communicate, and dynamically exchange control among themselves. It has two components for each participating site:

1. A *button* containing the name of the user as well as the host name to indicate who holds the token. The Token Holder is identified by highlighting its button by a green color whereas all other users have yellow buttons.
2. A *message box* for each user facilitates easy communication by allowing all users to simultaneously type in their respective message boxes. There are no possibilities of contentions here. This is similar to the *talk utility* available on Unix systems.

Thus, in our collaborative environment: (1) all users are allowed to interact with the message boxes for communication, and (2) only one user, the Token Holder, is allowed to interact with the entire application and relinquish the control by passing the token. We now explain some of the implementation details about the Display Controller and the FlowSynchronizer window.

Using Grab Command to Provide Control. Each participating user's interaction with the application is first limited to her own message box by using the Tcl command `grab`. Only the Token Holder's display screen does not have any `grab` command acting on any part of its application GUI. Then, as the token is passed among different users of the collaborating team, we dynamically change the effect of `grab` acting on each of the user's screen as follows: (1) release the `grab` for the message box of the user who is being given the control, (2) achieve a `grab` for the message box of the user who is giving up the control, and (3) retain the effect of `grab` on the message boxes of all the remaining users.

Achieving Concurrent Messaging. It is necessary for concurrent messaging that all users be able to type into message boxes simultaneously. Because of the default class bindings provided by Tk, a user has to bring the message box into focus before she can start typing. However, if a message box is brought into focus by clicking on it, it results in message boxes of other users to loose their focus, thereby re-directing their typed message into the new message box that is now in focus.

To overcome this problem, we first disable the default class binding for the message box and add a new binding script which automatically brings the message box into focus whenever the user's mouse cursor enters it. This ensures that every user has her message box in focus whenever their mouse cursor is over it.

Dynamic Code Generator (DCG). The functionality of the DCG is to intercept every Tcl/Tk command of the application code and modify it to generate a new Tcl code such that it will multi-cast the GUI of the application to '*n*' displays. We use a simple application `Hello World` to illustrate the implementation details of the DCG.

Figure 3(a) shows the original application code that creates and displays a label widget containing the text "Hello World" on the user's screen. The tcl code for multi-casting such a simple example to three display screens is shown in Figure 3(b). We first create a toplevel window on each user's display screen which acts as a place holder for the various widgets that will be created by the application code. The initialization script shows how to create such place holders on each display screen using the command `toplevel`. Next, for every command in the original application code that is related to GUI, the DCG generates three commands, one for each display screen. Thus, we have three `label` and three `pack` commands in the multi-casting example code. This code merely illustrates one method to broadcast the application GUI to three display screens and does not include scripts to initialize the FlowSynchronization window and its request-based control mechanism.

4 Session Recording and Playback Environment

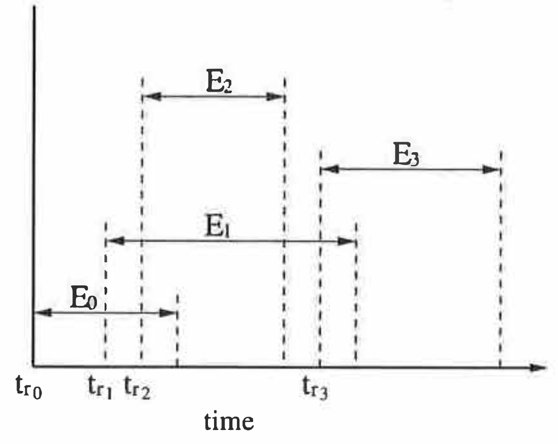
Recording and playback environment provides a mechanism of 'taking minutes', not only of the interactive discussions, but also of the menu-based commands associated with different tools in the workflow, of user-entered data inputs, and of user-queried data outputs. There are several benefits to the recording and playback mechanism:

- support for automated software documentation and tutorials, capturing the dynamics of software interactions for playback and review at a later time;
- study of activities and feedback on how teams actually collaborate, to improve the effectiveness and efficiency of collaborative environments;
- remote assistance, by selecting and playing back effective solutions recorded earlier;
- recording of solutions to frequently asked questions (FAQs) - these recordings can be played back at a later time on demand;
- automated performance evaluation of tcl applications, by repeating executions of pre-recorded sessions under different cpu loads and varying conditions of network traffic - such an example is described in the next section on experiments.

TkReplay [6] is Tcl/Tk program that allows one to record all user actions of Tk programs and then replay them. It uses the Tk command `send` to intercept each user action and saving it as a script file. This script file may then be reloaded and played back later to emulate the original user actions. The TkReplay will not work with Tcl/Tk applications in which the Tk command `send` is disabled. The `send` command is potentially a serious security loophole, since any application that can connect to your X server can send scripts to your applications. These incoming scripts can use Tcl to read and write your files and invoke subprocesses under your name. Hence, it is common for applications written in Tcl/Tk to disable the the operation of the `send` command. Our desktop environment is an example of one such Tcl/Tk application in which the `send` command has been disabled. Therefore, TkReplay is not suitable for use within our desktop. Instead, we make use of the Tk command `event generate` to playback the recorded events. We next define few terms before explaining the architecture of recording and playback mechanism.

Event is an occurrence of an interaction between the user and the windowing system. The windowing system constitutes of the local display, the keyboard, and the mouse. *Recording and playback* essentially involves capturing all events that are generated due to user actions during a session, and reproducing those events in exactly the same sequence as they were generated.

(a) Typical timing diagram of event generation



(b) Events and its timing information stored

Event list	Event details	Start times
E_0	...	t_{r_0}
E_1	...	t_{r_1}
⋮		⋮
E_n	...	t_{r_n}

Fig. 4. Details of event generation and timing.

Figure 4(a) shows generation of various events and its associated timing information. The X-axis depicts time increasing from left to right, whereas the length of each event shown represents the amount of time it takes to process an event after it has been generated. The terminology used in the illustrated timing diagram is as follows:

E_i	The i^{th} event in a session.
t_{r_i}	The instant at which the event E_i occurs.
$t_{r_{i+1}} - t_{r_i}$	The time difference between the occurrence of the event E_{i+1} and the event E_i.
n	The total number of events generated during a session.

We classify events generated into two categories: (1) *window events* are those generated as a direct result of the user's interaction with the application, and (2) *synthesized events* are those which emulate the window events using the Tcl/Tk commands from within the application program and are *not* a result of user actions.

Every window event consists of at least one primitive component. Examples of primitive components include: ButtonPress, ButtonRelease, MouseMotion, KeyPress, etc., and are used to identify the type of event that is occurring. However, additional information may be necessary to fully describe an event and is available as a secondary component. Secondary components represent details such as the x-y coordinates of the mouse cursor on the screen, the name of key that was pressed, the button number of the mouse that was clicked, etc. The primary and secondary components can then be used together to generate a synthesized event from within the Tcl application. Thus, Figure 4(b) shows a list of events generated and its primary and secondary components along with the timing information that is necessary to synthesize a window event.

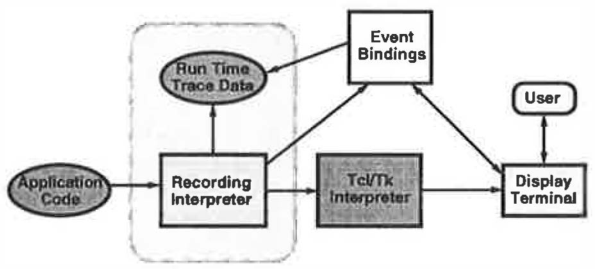


Fig. 5. Architecture of recording of a session.

Recording Session Architecture. Figure 5 shows the architecture of how to record an entire session consisting of all the user interactions with the Tcl application. It is necessary to intercept and capture the entire sequence of user actions and save it in a file as a *run time trace data*. This involves two steps:

1. A *recording interpreter* intercepts every Tcl/Tk command of the application that is being passed to the Tcl interpreter and dynamically associates a new class of binding called *RecordTaker* to every widget that is being created for the application. Thus, for example, if a button widget called '.b' is being created by the application, then we use the following command to associate the *RecordTaker* bind class with it:

```
bindtags .b [linsert [bindtags .b] 0 RecordTaker]
```

2. Whenever a user interacts with the Tcl application, the binding script for the *RecordTaker* is invoked first, before processing any of its default bindings. The binding script for *RecordTaker* is designed to capture all the information relevant to a specific widget and save it as the *run time trace data*. A typical run time trace data that is saved in a file is shown in Figure 4(b).

Playback Session Architecture. Figure 6 shows the architecture of playback of a recorded session which is stored in a file containing the *run time trace data*. The playback of a recorded session is initialized by invoking the original Tcl/Tk application code with the Tcl interpreter. The *trace data processor* then reads the *run time trace data* from the specified file and processes it to create synthesized events. The timing information associated with each event recorded is very important and critical in synchronization of the synthesized event during playback. It is possible to playback an exact replica of the recorded session, provided the cpu load is not significantly different from that during the recording process. The effect of variations in cpu load is as follows:

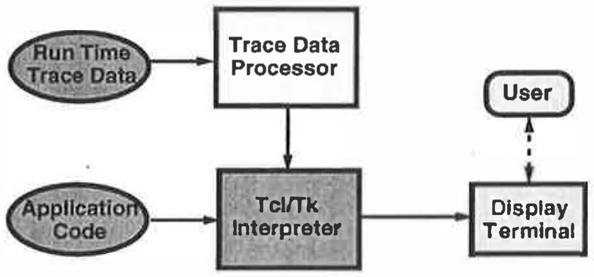


Fig. 6. Architecture of playback of a session.

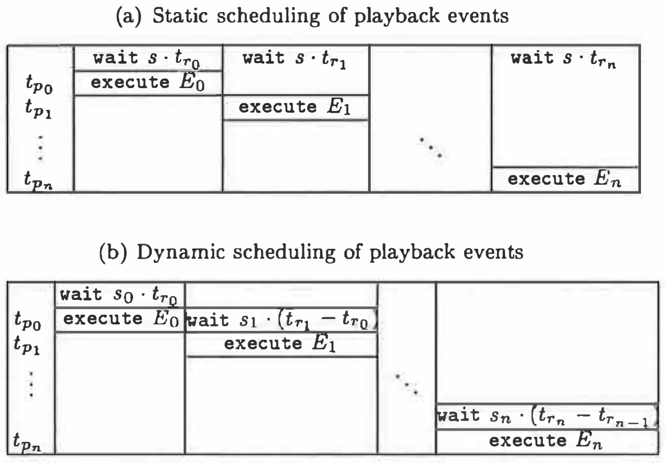
1. If the cpu load has increased during playback, then it will simply result in longer time for completion of the playback session.
2. On the other hand, if the cpu load has significantly decreased during playback, then some of the synthesized events may be generated even before processing of few of the earlier events has completed. This can result in generation of synthesized events which are not in original sequence and hence the playback session may fail.

Based on these experiences, we decided to allow the user to control the speed of execution during playback. We next describe the mechanism for scheduling synthesized events that facilitates the user to achieve varying speed of playback session.

Event Scheduling. We first define the terminologies used for scheduling the events of a playback session:

t_{p_i}	The time at which the synthesized event E_i is scheduled for playback.
s	The scaling factor which determines the speed of the entire playback session. It is constant for all the n events and is pre-computed before the commencement of a playback session.
s_i	This is a dynamic scaling factor for the i^{th} synthesized event. The value of this scaling factor is controlled by the user and can change during the playback session.

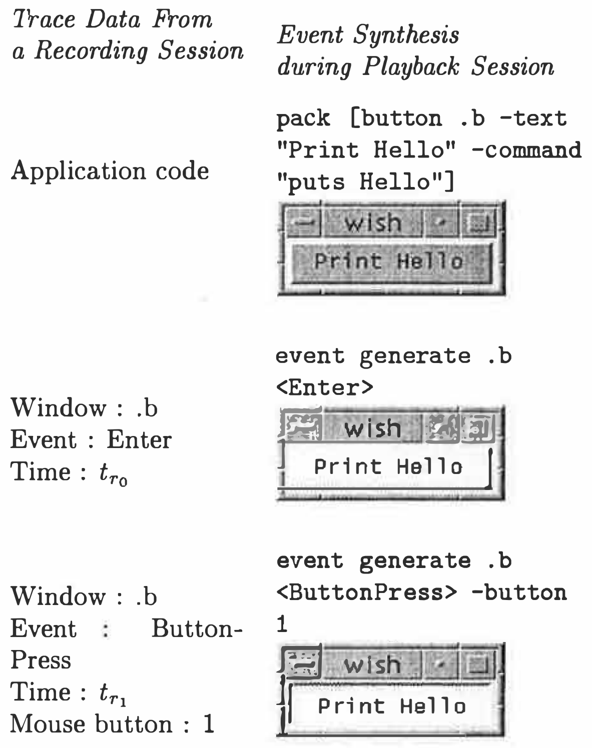
Figure 7(a) and (b) show two schemes of scheduling synthesized events. Both the schemes use the Tcl command `after` to generate event at a specified



(c) Comparision of static and dynamic scheduling of playback events

Playback time	Static scheduling	Dynamic scheduling
t_{p_0}	$s \cdot t_{r_0}$	$s_0 \cdot t_{r_0}$
t_{p_1}	$s \cdot t_{r_1}$	$t_{p_0} + s_1 \cdot (t_{r_1} - t_{r_0})$
t_{p_2}	$s \cdot t_{r_2}$	$t_{p_1} + s_2 \cdot (t_{r_2} - t_{r_1})$
$\vdots$	$\vdots$	$\vdots$
t_{p_n}	$s \cdot t_{r_3}$	$t_{pn-1} + s_n \cdot (t_{r_n} - t_{r_{n-1}})$

Fig. 7. Scheduling playback of recorded events.



Trace Data From a Recording Session | *Event Synthesis during Playback Session*

Application code

```
pack [button .b -text
"Print Hello" -command
"puts Hello"]
```

Window : .b
Event : Enter
Time : t_{r_0}

```
event generate .b
<Enter>
```

Window : .b
Event : Button-Press
Time : t_{r_1}
Mouse button : 1

```
event generate .b
<ButtonPress> -button
1
```

Fig. 8. Details of a recording and playback session.

instant. In the static scheme, all the n events are scheduled for playback as soon as the Tcl application is initialized. Therefore, the speed of the playback session has to be decided in advance and cannot be changed later on during actual playback. On the other hand, in the dynamic scheme, each event is scheduled for playback as soon as the idle time interval of its previous event is completed, as shown in Figure 7(b). At the end of time interval $s_0 \cdot t_{r0}$, it not only generates the event E_0, but also schedules the `wait` $s_1 \cdot (t_{r1} - t_{r0})$ interval for generating the next event E_1. This feature gives the user the flexibility to dynamically control the speed of playback execution by changing the value of the scale factor s_i. In addition, it is also possible to pause the execution of the playback session by merely defering the scheduling of the next event.

Implementation Example. We use a simple application `Print Hello` button in Figure 8 to illustrate the main ideas used to implement the recording and playback mechanism.

The left side of the figure shows the trace data, and the right side of the figure shows the Tcl/Tk commands used for synthesis of the recorded events and the user views as each event is synthesized. The following steps are necessary to invoke the command associated with the button widget:

Step 1. Initialize the application code for displaying the button widget with the command

```
pack [button .b -text "Print Hello" \
        -command "puts Hello"]
```

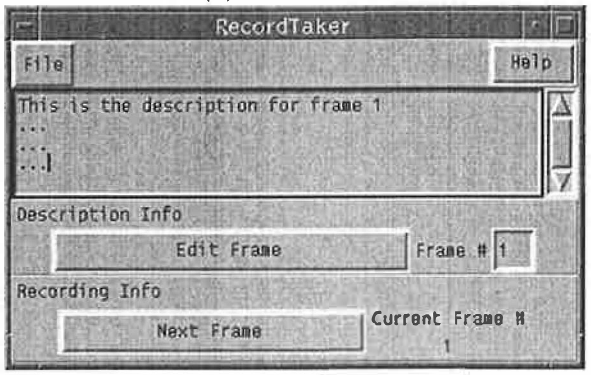
Step 2. It is necessary to activate the button widget before invoking its command. Therefore, synthesize the event 'Enter' in the window '.b' with the command `event generate .b <Enter>`.

Step 3. Next, synthesize the event 'ButtonPress' in the window '.b' with the command `event generate .b <ButtonPress> -button 1` This results in printing of the text "Hello" to the standard output.

Recording and Playback Tools. Figure 9 shows the GUI of two tools - a *RecordTaker* that assists users to create customized recordings, and a *PlaybackMaker* for playback of a recorded session with capabilities to control its speed or pause as desired. The *RecordTaker* also allow the users to record a session as a series of several smaller steps, called *frames*, instead of one single large recording. In addition, a user can type in appropriate text for each frame describing its functionality.

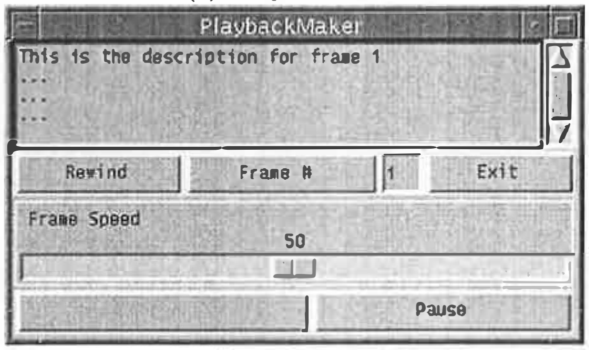
Thus, once a session is recorded in several frames, the *PlaybackMaker* automatically stops at the completion of each frame, displays the associated text for

(a) *RecordTaker*



This is the description for frame 1
...
...
...

(b) *PlaybackMaker*



This is the description for frame 1
...
...
...

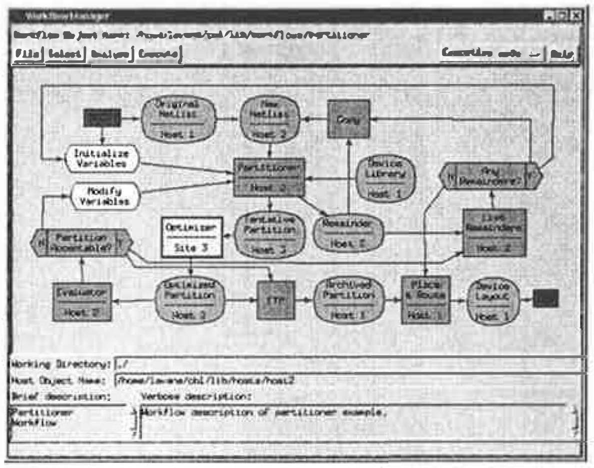
Fig. 9. Recording and playback tools.

the user to read and waits for the user to click on the 'Continue' button to start processing the next frame. This feature is very useful for recording tutorial sessions, where each tutorial consists of several frames with a text describing each frame during playback.

5 Experiments

The prototype of an environment that records, plays back and executes a Tcl/Tk collaborative Internet-based desktop, is being put to the test as an integral part of a national-level collaborative and distributed design project involving teams at 6 sites [12]. Specifically, the desktop brings together distributed data, application workflows, and teams into collaborative sessions that share the control of the desktop editing and execution. A typical workflow, such as the one shown in Figure 10, invokes distributed tools and data to support a major phase in the design of microelectronic systems. A detailed description is available in [8, 10].

We argue that recording and playback of collaborative user interactions can have a wide-range of applications, such as: 'keeping minutes' of interactive discussions, clicks of menu-specific commands associated with different tools on the shared desktop, user-entered data and control inputs, user-queried



Working Directory: ./
Host Object Name: /home/lavana/cbl/lib/hosts/host2
Brief description: Partitioner Workflow
Verbose description: Workflow description of partitioner example.

Fig. 10. Partitioner workflow.

data outputs, support for automated software documentation, tutorials, collaborative playback of tutorials and solutions recorded earlier, etc. The 540 experiments, summarized in this section, are the initial part of the Internet desktop environment performance and functionality evaluation, conducted before its release to Vela Project participants and others.

Each of these experiments relies on *interactive* user inputs. To maintain consistency of user inputs during the repeated trial executions across the Internet (with variable quality-of-service), we first *record* a single reference instance of each test case on the local server (without relying on the network) and then move these recordings to cross-state and cross-country servers on the Internet. Each server has an executable version of *ReubenDesktop*, *OmniBrowser*, *RecordTaker*, and *PlaybackMaker*. The experiments are initiated with a *playback* that executes recorded instances of test cases, multi-casting them to 1, 2, or 3 workstation displays at CBL. Additional details about these tools are available in [8, 9, 11]. Experiments reported in this section support a conjecture that will be the subject of more detailed experimentation later:

> *Task-specific performance of a single/multiple client-server* ReubenDesktop *execution can be predicted, under comparable server and network loading, by measuring the performance of pre-recorded task-specific experiments that are executed and multi-cast by the server to one/multiple client displays.*

In other words, to assess the performance of *interactive distributed sessions that involve one or more participants*, we have verified that the experiments, as reported in this section, can be extrapolated by measuring the performance of single- and multi-cast executions that are based on playback of pre-recorded experiments on a reference server. The benefits of not requiring a number of individuals to sit through repeated session experiments are obvious. Specifics about the testbed configurations, test cases considered, and graphical results follow.

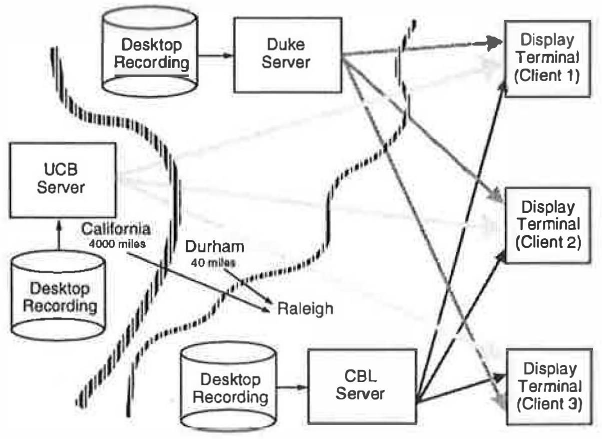


Fig. 11. Test-bed configuration for experiments.

Testbed Configurations. In order to approximate typical instances of a distributed multi-site collaborative desktop environment, we used a test-bed setup, as shown in Figure 11, to create:
(1) a local environment by installing the desktop software on a CBL server[1] at NC State University in Raleigh, NC, which is multi-casting its desktop to one or more CBL client hosts;
(2) a cross-state environment by installing the desktop software on a server[2] at Duke University in Durham, NC, which is multi-casting its desktop to one or more CBL client hosts; and
(3) a cross-country environment by installing the desktop software on a server[3] at the University of California in Berkeley, CA, which is multi-casting its desktop to one or more CBL client hosts.
We have carefully selected two remote servers such that the physical distance of approximately 40 miles and 4000 miles from Raleigh to Durham and Berkeley respectively, represents a realistic test-bed for performance evaluation. The arrows, shown in Figure 11, depict broadcasting of the desktop environment from each server to the three display terminals at CBL in Raleigh.
Test Cases. We have created and *recorded*, directly on the CBL server under negligible loading conditions, six test cases of collaborative sessions with useful attributes that demonstrate typical user-invoked tasks. The brief description that follows includes the reports of *real_time*, *user_time* and *system_time* as produced by the Unix utility `time`. The 'real_time' corresponds to the 'stopwatch_time' that could have been obtained by the user monitoring the task. The 'user_time' is the time required by the CPU to complete the task. The 'system_time' is the CPU time required by the system on behalf of the task. A brief description of all test cases engaging two participants, that were recorded for the experiment, follows.

[1]SUN SPARC 20 (chip=60MHz memory=64Mb swap=732Mb)
[2]SUN Ultra 1 (chip=167MHz memory=256Mb swap=288Mb)
[3]SUN SPARC 20 (chip=60MHz memory=96Mb swap=365Mb)

(1) Co-editing-1 (real_time=119.4s, user_time=31.1s, system_time=1.5s): Using *ReubenDesktop*, we *open, and edit*, a simple 4-node, 3-arc workflow by selecting, opening, and closing a single data file node-configuration window.
(2) Co-editing-2 (real_time=153.1s, user_time=44.0s, system_time=1.9s): Using *ReubenDesktop*, we *open, and edit*, the same 4-node, 3-arc workflow by selecting, opening, and closing a single data file node-configuration window and a single program node-configuration window.
(3) Co-editing-3 (real_time=223.8s, user_time=67.5s, system_time=2.5s): Using *ReubenDesktop*, we *open, and edit*, the 17 node, 22 arc workflow by selecting, opening, and closing 3 data files and a single program node-configuration window.
(4) Co-browsing-1 (real_time=136.7s, user_time=56.1s, system_time=2.1s): Using *OmniBrowser*, we *traverse* a directory structure, located on the server's local file system, across 3-levels, with up to 141 items in each directory. The directory structures of all the three servers were made exactly the same for uniform comparison.
(5) Co-browsing-2 (real_time=159.2s, user_time=97.5s, system_time=5.0s): Using *OmniBrowser*, we *select, open, and scroll*, from start to end, the same copy of a text file of about 1000 pages (2.2Mb), located on each server.
(6) Co-execution-1 (real_time=123.9s, user_time=90.0s, system_time=3.8s): Using *ReubenDesktop*, we *open, and execute*, the hierarchical workflow in Figure 10. As shown, the workflow has 22 nodes and 28 arcs; during execution, the node labeled as `optimizer` expands into a sub-workflow with 14 nodes and 15 arcs. All test cases involved two participants working collaboratively and consisted of exchanges of several dialogs via the *FlowSynchronizer* between the two, during each recording session.
Evaluation Method. All software and the files of six test cases, recorded directly on the CBL server, have been replicated on the server at Duke U. and the server at UCB. Scripts have been invoked, *during the night when both servers and the network were*

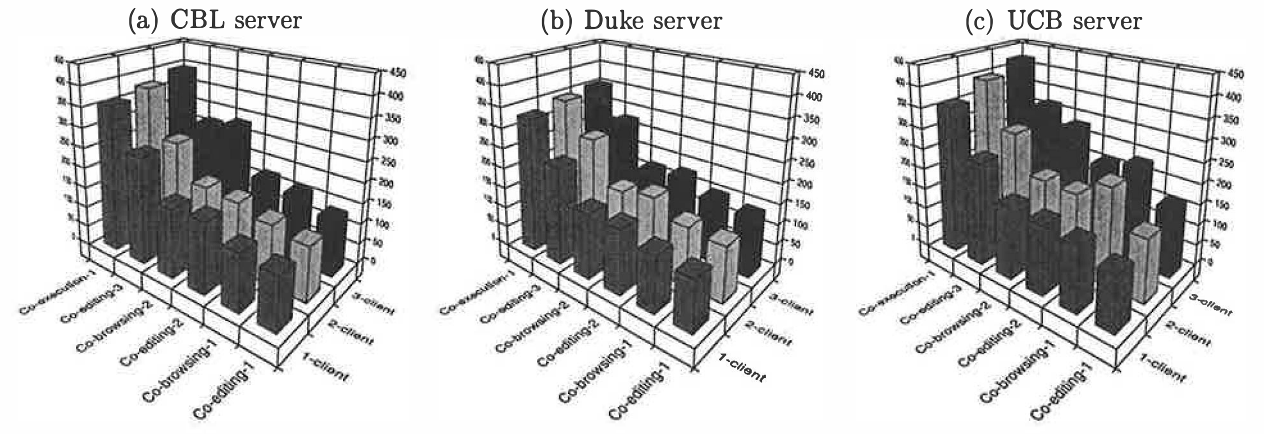


Fig. 12. Evaluation results of 540 experiments on three servers.

least loaded, to execute the 540 experiments as follows:

From each of the three servers, execute and multicast 10-times, with interval of 30 seconds between each execution:

(1) successively to one, two, and three client hosts at CBL, recordings of *co-editing-1, co-editing-2, co-editing-3*;

(2) successively to one, two, and three client hosts at CBL, recordings of *co-browsing-1, co-browsing-2*;

(3) successively to one, two, and three client hosts at CBL, recording of *co-execution-1.*

A log file, generated by `time` (real_time, user_time, system_time) command, archives timing data for each experiment. Similarly, a log file, generated by `sar` (system activity report) command, archives the load on each of the three servers during the execution of these experiments. The log file generated by `sar` provided the information whether or not both the load on the server and the network was sufficiently stable to accept the 'real_time' and 'user_time' results for tabulation.

The data generated as a result of evaluating the six test cases are tabulated and plotted as a 3-dimensional graphs shown in Figure 12 for each server. The x-axis in each plot lists the name of the test case and the y-axis represents the number of clients that a specific test case was multi-casted to. The time required for execution of each test case with 1-, 2- and 3-clients is then represented as a bar on the z-axis.

Summary of Results. The data plotted in Figure 12 allows us to evaluate the performance of Internet-based desktop environments.

1. The 'real_time' for playback to a single-client on the reference server is approximately the same as the time required to record the test cases.

2. The 'real_time' for playback from other servers varies, depending on the distance between the host server and its clients and the characteristics of the host server. Specifically, for single-client playback, Duke server consistently reported least execution times, followed by CBL server and UCB server. This is attributed to the higher performance server at Duke. However, for multi-clients, the execution times increased with distance in the order CBL, Duke, and UCB.

3. When the experiment is multi-cast to 2-clients or 3-clients, it takes slightly more time, on the order of a few seconds, for execution than the time required for single client execution. The negligible increase in the playback time for multi-client execution is due to the fact that the exchange of dialog among participants is computationally least intensive.

4. The variations in minimum and maximum values of 'real_time' for each experiment are negligible since the experiments were performed during the night. However, the same experiments showed significant variations during the day when the network traffic and the server load are unpredictable.

5. Comparing the 'user_time' and the 'system_time' for each server, we find that the CBL server requires the most CPU time and the Duke server requires the least CPU time. This follows directly from the different types of processors and the configuration of each server.

Observations. The successful completion of all 540 experiments provides us with assurance that the experiments are consistently reproducible on a variety of servers, given that the server nominal load is small and that the network is stable. Specifically, we confirmed that

• Repeated *real time* executions of experiments, where user-inputs are carefully and consistently entered (rather than pre-recorded), gives 'real_time', 'user_time', and 'system_time' performance that is comparable (within 10%) to the times reported for pre-recorded execution on any server – provided that the server load and network conditions are as favorable.

• The performance of the Internet-based desktop environment, even in a *collaborative mode*, is quite good under nominal network traffic and load on the server. Hence, with sufficient network bandwidth and powerful processors, it is possible to work collaboratively with efficiency and effectiveness even when participants are dispersed across the continent.

• As the number of clients, corresponding to each participant, increase from 1 to n during playback, the increase in 'real_time' execution is on the order of a few seconds only. Again, this increase is subject to the server and network performance and the amount of dialog among participants present in the recording.

6 Software Availability and Status

The collaboration, recording and playback features are all currently integrated into a single desktop environment. We are planning to unbundle the desktop environment and make them available as separate packages:

(1) CollabTclTk. Collaboration of *any* tcl-application to multiple sites,

(2) RecordnPlayTclTk. Recording and playback session of *any* tcl-application, and

(3) ReubenDesktop. Internet-based desktops integrated with collaboration and recording/playback packages.

For more details about the current status of these packages, please visit:

`http://www.cbl.ncsu.edu/software`

7 Conclusions and Future Work

We have introduced a new paradigm and a prototype implementation of a collaborative and recordable environment on the Internet using Tcl/Tk. Complementing the objectives of the user-reconfigurable Internet-based desktop environment, this environment supports

• *peer-to-peer* interaction between members of any team;

• *peer-to-workflow* interaction between any team member to any object in the workflow;

• *recording and playback* of interactive execution of Tcl/Tk applications and collaborative sessions.

Future work on collaborative and recordable environments should address the following issues:

Security of collaboration. Our current implementation is 'insecure' since it relies on participating hosts to open their X displays for remote connection. Users who are behind company 'firewalls' are not able to participate, given the present set-up.

Dynamic collaborative team. Currently, we have to close and restart the application for changing the number of participants. In the future, we want to support dynamic addition and removal of users participating in collaboration.

Editing of the recording session. Recording, *without errors*, of a long complex session is difficult. Hence, recording sessions of shorter durations and slicing them together later for playback would be more effective.

ACKNOWLEDGMENTS. We could not have reported as comprehensively on the results of our Internet Desktop experiments without getting generous user accounts on two remote servers, facilitated by Dr. Richard Newton at UC Berkeley and Dr. Gershon Kedem at Duke U. We thank them and their support staff for this privilege.

References

[1] Scriptics Corporation. Published under URL `http://www.scriptics.com/`, 1998.

[2] The Tcl/Tk Consortium. Published under URL `http://www.tclconsortium.org/`, 1998.

[3] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[4] M. Rosenman and S. Greenberg. Designing Real-Time Groupware with GroupKit, A Groupware Toolkit. In *ACM Transactions on Computer Human Interaction*, 1996.

[5] GroupKit. Published under URL `http://www.cpsc.ucalgary.ca/grouplab/groupkit`, 1997.

[6] Charles Crowley. TkReplay: Record and Replay in Tk. In *Proceedings of the Tcl/Tk Workshop, Toronto, Canada*, July 1995.

[7] D. Libes. *Exploring Expect*. O'Reilly and Associates, 1995.

[8] H. Lavana, A. Khetawat, F. Brglez, and K. Kozminski. Executable Workflows: A Paradigm for Collaborative Design on the Internet. In *Proceedings of the 34th Design Automation Conference*, pages 553–558, June 1997. Also available at `http://www.cbl.ncsu.edu/publications/-#1997-DAC-Lavana`.

[9] Amit Khetawat. Collaborative Computing on the Internet. Master's thesis, Electrical and Computer Engineering, North Carolina State University, Raleigh, N.C., May 1997. Also available at `http://www.cbl.ncsu.edu/-publications/#1997-Thesis-MS-Khetawat`.

[10] H. Lavana, A. Khetawat, and F. Brglez. Internet-based Workflows: A Paradigm for Dynamically Reconfigurable Desktop Environments. In *ACM Proceedings of the International Conference on Supporting Group Work*, Nov 1997. Also available at `http://www.cbl.ncsu.edu/-publications/#1997-GROUP-Lavana`.

[11] H. Lavana, A. Khetawat, and F. Brglez. `REUBEN` 1.0 User's Guide. CBL, Research IV, NCSU Centennial Campus, Box 7550, Raleigh, NC 27695, 1998. To be available at `http://www.cbl.ncsu.edu/publications/`.

[12] Globally distributed microsystem design: Proof-of-concept. A university-based project involving teams at 6 sites. See the project home page at `http://www.cbl.ncsu.edu/vela` for more details.

Creating A Multimedia Extension for Tcl Using the Java Media Framework

Moses DeJong, Brian Bailey, and Joseph A. Konstan
University of Minnesota
Computer Science and Engineering Department
{dejong, bailey, konstan}@cs.umn.edu

Abstract

As multimedia capable computers become cheaper and more pervasive in the consumer and corporate markets, and as the availability of digital information increases, the need for low-cost, cross-platform multimedia applications will steadily rise. However, because Tcl lacks native support for continuous media streams, such as audio, video, and animation, it is not well suited for this emerging application domain. At the same time, Java now provides a set of class libraries, called the Java Media Framework (JMF), which provides the multimedia support that Tcl lacks. With the recently introduced integration of Tcl and Java, Java can now be used to provide the cross-platform multimedia support required by Tcl; whereas Tcl can be used to provide the easy-to-use programming environment required for building multimedia applications. In this paper, we introduce a Tcl extension that provides a high-level scripting interface to the Java Media Framework. In addition, we will highlight some interesting problems in the current Tcl/Java package as well as suggest some potential solutions. This paper will benefit Tcl programmers who would like to learn more about using Tcl to build multimedia applications, integrating Tcl and Java, or the multimedia support provided by the JMF.

Keywords

Multimedia, Tcl extension, Synchronization, Jacl, TclBlend, Java, Java Media Framework

1 Introduction

Tcl [9] has never directly supported continuous media streams, such as audio, video, or animations. Tcl programmers wanting to include these types of media objects in order to create desktop conferencing applications, electronic manuals, voice activated interfaces, or even simple multimedia presentations, have had to rely on fellow Tcl developers to build, distribute, and support multimedia extensions. Due to the huge variety of media formats, transport protocols, playout devices, and synchronization models in use today, it is no wonder the core Tcl group has shied away from attempting to directly provide multimedia support. One of the more popular multimedia extensions for Tcl is the Continuous Media Toolkit (CMT) [10] developed at California-Berkeley by the Plateau project group. CMT is a flexible, low-level multimedia toolkit supporting a variety of media formats, transport protocols, and playout devices. Although CMT has proven to be very useful in our research, it suffers from the same problems as other Tcl extensions:

- *Portability.* Designing extensions that are portable across multiple flavors of Unix, Windows, and the Macintosh is a non-trivial task. In fact, most extensions available today, including CMT, do not provide this level of cross-platform portability.
- *Compatibility.* With each new release of the Tcl/Tk core, an extension developer must update and test their extension to ensure that it still functions properly. Unfortunately, this has not been easy as Tcl/Tk has changed substantially in at least the following core areas; file I/O, image support, sockets, event handling, and object APIs, to name just a few. For some extensions, such as CMT, this has caused a lag of at least 6 months behind new releases of Tcl/Tk.
- *Configurability.* Extensions cannot easily access the configuration information Tcl uses for its own compilation. Extension writers must effectively recreate this same information, which is why almost every Tcl extension comes with its own set of configuration scripts.
- *Dependencies.* Many Tcl extensions are not standalone; i.e., they require other extensions in order to function properly. For example, CMT requires the Tcl-Dp [12] extension in order to leverage distributed services. As the number of extension dependencies increases, so does the difficulty of upgrading an extension to support the latest release of Tcl/Tk.

Within the last year, JavaSoft has introduced the Java Media Framework (JMF) [4]. The JMF is a Java [1] package that provides support for the local playback of a wide variety of audio (AIFF, AU, DVI, MIDI, and MPEG-1), video (H.261, H.263, MJPEG, MPEG-1, AVI, QuickTime), and animation formats (Apple Animation). As the JMF evolves, it promises to also support audio recording, audio mixing, video capture, streamed playback, and emerging multimedia standards such as MPEG-4. The JMF already provides roughly the same functionality offered by CMT, but without many of the problems inherent in Tcl extensions as described above. Thus, the JMF deserves serious consideration as the underlying toolkit used in commercial multimedia applications or even research projects. However, building multimedia applications in pure Java code using the JMF has several drawbacks:

- *Complexity.* The JMF currently supports only primitive media events such as notification of a playback rate change. Exactly how these events affect the rest of the application must be defined by the developer in low level Java code.
- *Lack of higher-level synchronization models.* Without higher-level synchronization support, building complex multimedia applications becomes difficult. This is roughly equivalent to building user interfaces by invoking low level X routines instead of using a higher-level toolkit like Tk.
- *Edit/Compile/Debug cycle.* Strongly typed, compiled languages like Java are great for developing low level systems, but they make poor tools for rapid prototyping and development of high-level multimedia applications.

Because of these drawbacks, we argue that the JMF would benefit from a scripting interface that would increase the usability and decrease the development effort required to build complex multimedia applications. Thus, the goal of this project was to use the Tcl/Java package [5, 6, 13] to provide a scripting interface to the JMF called **TJMF (Tcl/Java Media Framework)**. This paper describes our experiences and lessons learned from building this Tcl interface to the JMF. In section 2, a general overview of the features already provided by the Tcl/Java package is presented. Section 3 outlines the utility packages that provide the foundation for the TJMF extension. In Section 4, an example of using the TJMF extension to playback a MPEG movie as well as a detailed discussion of the extension is provided. In Section 5, we reflect on our experiences and lessons learned from using the Tcl/Java package. In the remaining sections of the paper, we present our conclusions and the results of recent work to improve the Tcl/Java package.

2 The Tcl/Java Package

To better integrate Tcl and Java, several key technologies have been developed. Jacl [6] is an implementation of a Tcl interpreter written entirely in the Java language. Jacl currently provides most of the functionality of the Tcl 8.0 interpreter. TclBlend [5] provides a native code library that accesses the Java Virtual Machine (JVM) using the Java Native Interface (JNI). The Java Package [5] is a set of Tcl commands providing access to the Java Reflection system. The Java Package provides commands that allow a Tcl interpreter to both allocate and invoke methods on Java objects. Tcl code using these commands will run seamlessly in either the TclBlend or Jacl implementations. In addition, extensions written in Java using the Java Package can be accessed in either Jacl or TclBlend, and are instantly portable to a wide variety of systems without configuration or recompilation. For the purposes of this paper, we use the term Tcl/Java package to collectively refer to Jacl, TclBlend, and the Java Package.

3 Integration Utilities

Before describing our multimedia extension for the Java Media Framework in detail, several supplemental packages need to be introduced. These utility packages provide the underlying support required by our multimedia extension and consist of the following:

- *Object-oriented support.* The lack of object-oriented support in Tcl makes modular development difficult. For several reasons that will be explained shortly, none of the current object-oriented extensions were usable in this project, and thus a Tcl-only object-oriented package was created.
- *Event mapping system.* A package to map Java events to Tcl callbacks was created so that events from the JMF components could be managed in user-defined Tcl code.
- *Pack geometry manager for the Java AWT.* The existing Java layout managers are inflexible and difficult to use. In response, a new layout manager, based upon the pack geometry manager of Tk, was created for Java AWT components.
- *Package command for Jacl.* Initial versions of Jacl did not provide an implementation of the Tcl

`package` command, but since our project required this functionality, the Jacl interpreter was extended with an equivalent Java implementation.

3.1 Object-Oriented Support

From our experience, the most severe impediment to modular development in Tcl is the lack of object-oriented support. Tcl has no built-in support for aggregate data types, other than lists and arrays, and also lacks support for encapsulation of data. [incr Tcl][8], an extension that provides object-oriented support, could not be used in this project for two reasons. First, [incr Tcl] requires patches to the Tcl core and its own C level libraries, and thus cannot be run in Jacl. Second, TclBlend requires the use of Tcl8.0 which is not yet supported by [incr Tcl]. Although several Tcl-only object-oriented packages exist within the Tcl community, we could find none that would seamlessly work with both Tcl8.0 and Jacl. In one case, the use of "::" within global procedure names conflicted with the new namespace feature of Tcl8.0. Because of these problems, we designed a simple, Tcl-only, object-oriented package that is compatible with all current Tcl releases, including Jacl. The package provides basic support for classes, encapsulation, and composition.

3.2 Event Mapping System

In order to script JMF components using Tcl, Java events need to be mapped to Tcl callback procedures. This type of mapping is similar to the way X events are mapped to Tcl procedures using the `bind` command from Tk. Such functionality would allow a Tcl programmer to create JMF components; e.g., a MPEG player along with a set of VCR controls, and manage the component interaction in Tcl code. The `java::bind` command included in the Tcl/Java package already provides this type of Java to Tcl event mapping, but could not be used because it only supports JavaBean events, which the JMF does not generate. In our TJMF extension, a Java class receives JMF events, translates them into a descriptive string, and invokes the appropriate Tcl callback. Example 1 demonstrates how a Tcl callback would be invoked when a Java event is received from the JMF.

3.3 Pack Geometry Manager

Java's Abstract Windowing Toolkit (AWT) provides poor support for geometry management. The default layout managers are lacking in functionality, error prone, and difficult to use. On the other hand, Tk's pack geometry manager is powerful, easy to use, and allows for rapid prototyping of user interfaces. In order to gain these same advantages when developing interfaces with the AWT, a pack geometry manager was created by porting the packer layout algorithm to the AWT 1.1 LayoutManager interface. Java programmers can use this layout manager to gain pack functionality for AWT components. Although this functionality could be accessed from Tcl using the Tcl/Java package, the layout manager would be easier to use if wrapped in a Tcl command. For this reason, a `jtk::pack` command was created that accepts the same options as the `pack` command of Tk. Use of this command can be seen in Example 1.

3.4 Package Command

Tcl's `package` command provides a convenient means for managing logically connected source files within an application. Without it, managing which source files have already been loaded into the interpreter can become very complex. Unfortunately, early versions of Jacl did not support the `package` command. Initially, the Tcl/Java group was contacted, but since they could not begin working on unimplemented features until after the 1.0 release date, the decision was made to implement the `package` command ourselves. The implementation was performed by translating the source code from the C version of the `package` command into Java source code. The Java version of the command was then checked by running the regression tests from the main Tcl distribution. Once stabilized, the code was sent to the Tcl/Java group and has since been included in subsequent releases of Jacl.

These packages provide the foundation on which the TJMF extension is built. With each of these packages in place, a detailed explanation of the TJMF extension will now be provided.

4 The TJMF Extension

The motivation behind the TJMF extension began while assisting another research project in the University of Minnesota's kinesiology department. The kinesiology department was conducting research involving a sports performance measuring system that required a multimedia interface. While a number of multimedia technologies are available, the decision was made to use Java and the JMF. Writing Java code requires much less time than a more complex language like C or C++, but development was still slow due to the edit/compile/debug cycle; especially because restarting the application from scratch took a long time. Although the project was a success, the implementation effort could have been reduced by eliminating the need to continually recompile and restart the application. Having used multimedia scripting tools in the past, it was clear that application developers using the JMF would benefit from a scripting front-end. Thus, the idea of providing a Tcl scripting interface to the JMF was conceived.

4.1 TJMF Usage

The interface to our TJMF extension was designed with two goals in mind. First, Tcl programmers should be able to use the TJMF with minimal effort. Second, Java programmers already familiar with the JMF, but who desire a faster development cycle, could use the TJMF without learning a whole new multimedia toolkit. This should help convince Java programmers using the JMF to switch to Tcl and the TJMF. Example 1 demonstrates how a MPEG file would be displayed with the TJMF extension.

Example 1

```
#Add TJMF packages to this interpreter
package require Media

#Create a MPEG player
set player \
[MediaPlayer .play file:/C:/mpegs/movie.mpg]

#Create the event consumer object
set con [Consumer .con]

#Subscribe to events produced by MPEG player
$con subscribe $play

#Register an event handler
$con handler Media.PrefetchComplete \
  ready_callback

#Allocate Java Frame (toplevel in Tk)
set frame [java::new java.awt.Frame]
$frame setSize 300 300

#Callback for Media.PrefetchComplete event
proc ready_callback { player } {
  global frame

  #Get window that MPEG will be displayed in
  set child [$player getVisual]

  #Pack MPEG video window in the Frame
  jtk::pack $child -in $frame -padx 10

  #Map the Frame object
  $frame show

  #Start playback of the MPEG video
  $player start
}

#Start fetching MPEG data
#Generates Media.PrefetchComplete when done
$play prefetch
```

4.2 MediaPlayer Implementation

The majority of the TJMF functionality is contained within the MediaPlayer class. The MediaPlayer class can be used to playback a number of different media types such as audio, video, and animation. This class is implemented with our object-oriented package and provides an API that programmers can use to manipulate the underlying media streams. The MediaPlayer class encapsulates three key components that work together to provide the functionality of the TJMF extension.

The first component is an instance of the JMF class `javax.media.Player`. An instance of this class is created by invoking the Java method `javax.media.Manager.createPlayer()` and passing in the URL argument supplied to the MediaPlayer constructor. The MediaPlayer constructor returns a command procedure used to access that particular instance. When this command is invoked, the MediaPlayer translates it into the corresponding method invocation on the `javax.media.Player` object.

The second component encapsulated by the MediaPlayer is a Java class called an EventTranslator. The EventTranslator class implements the `javax.media.ControllerListener` interface so that it is able to receive events generated by the JMF. The EventTranslator is responsible for receiving a JMF event, and then translating it into a Tcl string that represents the event. For example, if the JMF generates a `javax.media.StopEvent` then the EventTranslator object will output the string `Media.Stop`. This output string is then used by the third component to generate a Tcl event.

The third component of the MediaPlayer is an instance of the Producer class which was implemented in Tcl code with our object-oriented package. When the JMF generates an event, the MediaPlayer uses the output of the EventTranslator as the input to the Producer object. The Producer contains a table that uses the event string to lookup a list of Consumer objects registered for that particular event. Each consumer registered for the event will then invoke its own user-defined callback.

Together, these three components make up the MediaPlayer implementation and define the process by which JMF events trigger user-defined Tcl callbacks.

4.3 Designing a Thread Safe Extension

When implementing the TJMF extension, one issue that needed to be addressed was that the JMF creates an additional thread to fetch, decode, and display the specified media stream. This separate thread generates asynchronous state update events which are received by the EventTanslator of the MediaPlayer (see section 4.2). After the event is translated, a procedure will need to be evaluated within the Tcl interpreter. However, because the Tcl interpreter is executing in a different thread, calling `interp.eval()` from the JMF thread could corrupt the interpreter. The problem is that the `interp.eval()` method is not thread-safe. The only safe way to evaluate a Tcl command from a separate thread is to place that command into a thread-safe Tcl event queue. To do this, the synchronized method `interp.getNotifier().queueEvent()` should be invoked. This method uses the synchronization primitives provided by the JVM to ensure thread-safety within the Tcl event queue. The Tcl interpreter will then remove the event object from the queue and invoke its `processEvent()` method from the interpreter's main thread. Example 2 demonstrates how to safely evaluate a procedure from a separate thread.

Example 2

```
String cmd = "string length hello";
EventProcessor ep = new EventProcessor(cmd);
interp.getNotifier().queueEvent(ep,
TCL.QUEUE_TAIL);

class EventProcessor extends TclEvent {
  private String script;
  EventProcessor(String s) {script = s;}

  public int processEvent(int flags) {
    try {
      interp.eval(script,flags);
    } catch (TclException e) {}
    return 1;
  }
}
```

4.4 Adding Higher Level Synchronization Support

Multimedia presentations are a common type of application that programmers may want to create using the TJMF extension. However, building these types of applications is not simple as several different approaches exist [2]. The Nsync toolkit [2] simplifies the creation of interactive multimedia presentations by providing a high-level, declarative interface. To increase the usability of the TJMF extension, we have produced a small compatibility layer so that the Nsync toolkit can be used on top of TJMF. By using the Nsync APIs, underlying details of the TJMF extension are hidden from the application programmer. We believe this adequately demonstrates the flexibility of the TJMF extension as scripts that were originally written for Nsync and CMT can be used with TJMF without significant modification.

5 Lessons Learned

In this section, we reflect on some of the issues faced during the development of the TJMF extension. While the Tcl/Java package has many useful features, it contains some problems that made development of the extension more difficult than it needed to be. The most serious problem in the Tcl/Java package is a design flaw that discards a Java object's type when reflected inside a Tcl interpreter. Java method invocation is also more difficult than is should be because the Java method resolver used in the Tcl/Java package is currently lacking needed functionality (see Section 5.2). In addition, problems exist when mapping some Tcl constructs to Java code and vice versa. These problem areas will be illustrated in the following subsections through a number of source code examples along with suggestions for improving the Tcl/Java package.

5.1 Java Objects Have No Class

Within the Tcl/Java package, all Java objects reflected in the Tcl interpreter lose their type information. Type information for Tcl objects is unimportant because "everything is a string"; however, this type information *is* important to the proper use of Java objects. Consider an example involving polymorphism. In Java, polymorphism is expressed through inheritance and allows a single object to be referenced as two distinct types (classes). In this situation, the type of the reference determines which methods can be invoked on the object. As shown in Example 3, the current Tcl/Java package discards the returned object's type by incorrectly casting it to the most derived type. By doing so, the user can now invoke methods that would ordinarily not be permitted.

Example 3

```
public class Hashtable2 extends
java.util.Hashtable
{
  public static java.util.Hashtable get() {
    return new Hashtable2();
  }
  public String foo() {
    return "CALLED";
  }
}

% set h [java::call Hashtable2 get]
% java::info class $h
Hashtable2
% $h foo
CALLED
```

This example demonstrates that the type information for the Java object is lost when it is reflected inside a Tcl interpreter. The type of the returned Java object should have been `java.util.Hashtable`, not `Hashtable2`. To fix this problem, the Tcl/Java reflection system must record the class (type) information for each Java object referenced. This modification would also allow the use of Java *interface* classes, which are currently inaccessible. To completely resolve this issue, a command to cast a Java object from one type to another would need to be included. We suggest adding a `java::cast` command as described in Example 4.

Example 4

```
#using Hashtable2 class from Example 3
% set h [java::call Hashtable2 get]
% java::info class $h
java.util.Hashtable
% $h foo
no such method "foo" in class java.util.
Hashtable
% set h2 [java::cast Hashtable2 $h]
% $h2 foo
CALLED
```

The behavior demonstrated in this example is consistent with common object-oriented principles.

5.2 The Java Method Resolver

In Java, *overloading* occurs when two or more methods have the same name but differ in the number and/or types of arguments. In order to distinguish among overloaded methods, a resolution algorithm must be applied to the argument types. Applying this algorithm is the responsibility of the Java method resolver.

When Tcl code makes a call to an overloaded Java method, the Java method resolver is invoked. In order to function properly, the Java method resolver must know the actual types of the arguments being passed. This information must be explicitly passed by the Tcl programmer using a special syntax. However, continually specifying method argument types quickly becomes tedious and error-prone. The Java method resolver is supposed to employ heuristics in order to disambiguate overloaded methods [6]. However, as demonstrated in Example 5, these heuristics are not employed.

Example 5

```
public class A {
  void foo(int i) {}
  void foo(String i) {}
}

% set obj [java::new A]
% $obj foo 1
ambiguous method signature "foo"
% $obj foo abcd
ambiguous method signature "foo"
```

The method resolver ignores Java object type information that could have been used to disambiguate the method signature. In fact, the reflection system already has this type information stored internally, it is simply ignored by the resolver. Example 6 demonstrates this problem.

Example 6

```
public class B {
  public void foo(java.util.Hashtable h) {}
  public void foo(java.util.Vector v) {}
}

% set hash [java::new java.util.Hashtable]
% set B [java::new B]
% $B foo $hash
ambiguous method signature "foo"
```

The only way to guarantee proper behavior by the resolver is to fully qualify each parameter of the Java method invocation. Using built-in Java objects, the Tcl programmer would have to specify object names like `java.io.File` or `java.util.zip.ZipFile`, which is annoying but still doable. If one starts working with classes that have longer names like `com.sun.java.swing.plaf.ToggleButtonUI`, the fully qualified syntax will quickly become tedious and error prone.

5.3 Parameter Specification

Within the Tcl/Java package, method invocations must be specified using the following format:

```
{method_name f1,f2,...,fn} a1,a2,...,an
```

where the f's represent the formal argument types and the a's represent the actual argument objects. The problem with this format is that every parameter type must be specified, regardless if that parameter type is required to resolve the method invocation. To alleviate this requirement, a more Java-like argument type specification could be used. Example 7 demonstrates an approach where only the parameter types needed to distinguish one method from another are supplied.

Example 7

```
public class C {
  public void foo (
   java.util.Hashtable h,
   java.util.Vector v,
   java.io.File f) {}

  public void foo(
    java.util.Hashtable h,
    java.util.Stack s,
    java.io.File f) {}
}

#This example assumes that h is a Hashtable
#v is a Vector, and f is a File object
% set C [java::new C]                    (a)
% $C {foo java.util.Hashtable            (b)
    java.util.Vector java.io.File} $h $v $f
% $C foo $h (java.util.Vector) $v $f     (c)
```

The second command above **(b)** shows the required method invocation format for the current Tcl/Java package. The third command above **(c)** shows the suggested notation in which only the necessary parameter types are specified.

5.4 Exceptional Problems

Exception handling is a widely used mechanism for raising, catching, and handling unexpected run-time conditions. Both Tcl and Java support similar exception handling mechanisms; however, a serious distinction exists. In Java, unlike Tcl, more than one type of exception can be thrown, and subsequently caught based on its type. In Tcl, this is not possible. As a result, handling Java exceptions in Tcl code is more difficult than it would be in Java code. Example 8 demonstrates the problem.

Example 8

```
public class D {
  public static void foo()
    throws NumberFormatException,
    java.io.IOException {
    //throw one of the exceptions
  }
}

//Handling the exceptions in Java code
public class E {
  public static void callfoo()
    throws NumberFormatException {
    try {
      D.foo();
    } catch (java.io.IOException e) {
      System.err.println(
        "Fatal : IoException");
      System.exit(-1);
    }
  }
}
```

```
#Handling the exceptions in Tcl code
if [catch {
  java::call D foo
} err] {
  if {[string match "*java.io.IOException*"
  $err]} {
    puts stderr "Fatal : IoException"
    exit -1
  }
  if {[string match
"*java.lang.NumberFormatException*" $err]} {
    global errorCode
    java::throw [lindex $errorCode 1]
  }
}
```

As shown in Example 8, Java exceptions are difficult to manage in Tcl code because Tcl inherently recognizes only a single type of exception. The situation becomes even more difficult when Tcl code needs to catch Tcl errors as well as Java exceptions. Every possible Java exception would need to be handled individually. Also the Tcl `catch` command does not provide a means to do local cleanup of resources without actually catching the exceptional condition. Java provides this functionality with the `try-catch-finally` construct. A `java::try` command would address this problem, but a better long-term solution would be to replace the Tcl `catch` and `error` commands with similar commands that are able to manage multiple exception conditions.

5.5 Combining Tk And AWT

At some point in the future, the Tcl/Java package must address the integration of the Java AWT and Tk. Currently, support for putting Tk windows inside Java windows, or vice versa, does not exist. This kind of support was described in [5], but has yet to be implemented.

5.6 Jacl

During this project, we discovered several problems with Jacl. First, run-time performance is poor. Jacl does not have a compiler like its C counterpart, so each command must be re-evaluated for every invocation, and Java code is generally about 10 times slower than C code. Second, Jacl can not currently be run inside of web browsers. The reason is that Jacl requires Java Reflection support which is not yet properly implemented in web browsers. Also, Jacl defines its own class loader which violates the default security policy for applets. Finally, Jacl contains a number of unimplemented, or only partially implemented commands, such as `socket`, `exec`, and `namespace`.

6 Conclusion

The TJMF extension is a powerful tool for adding multimedia capabilities to the Tcl programming language, without the inherent problems associated with C extensions. This was accomplished by using the Tcl/Java package to provide a scripting interface to the Java Media Framework. The TJMF extension was constructed with a liberal mix of Java and Tcl code:

Table 1: Lines of Tcl and Java code in the TJMF

Component	Tcl code	Java code
Media package	1000	500
Object package	1700	0
Event package	1300	0
Pack layout	100	1100
Other utilities	500	0
Total	**4600**	**1600**

The total implementation effort produced about 6200 lines of combined Java and Tcl code and was accomplished over a period of about 6 months. Each of the packages described in this paper are available for downloading at

http://www.cs.umn.edu/~dejong/tcl

Also, the object type reflection system outlined in section 5.1 and some of the method resolver heuristics described in section 5.2 have been implemented. These improvements will be made publicly available so that others can make better use of the Tcl/Java package.

By integrating Tcl and Java, each language can be used to do what it does best. Java can be used to provide the functional low-level components, while Tcl can be used to build applications by "gluing", or scripting these components together. The integration of Tcl and Java produces functionality that is beyond what could be achieved by using either language alone. With a few simple improvements outlined in this paper, the integration would become even more powerful and easier to use.

While much software has been created for Tcl/Tk, we have yet to see a solution to the portability problems inherent in using C as an extension language for Tcl. With the arrival of the Tcl/Java package, we hope that C extensions will become a thing of the past as the number of portable extensions for both Tcl and Java increase.

7 Acknowledgments

We are grateful for the help provided by the entire Tcl/Java team. Bryan Surles was particularly helpful in providing us with information about the proper use of the Tcl/Java package APIs. We would also like to thank Ray Johnson, Scott Stanton, Melissa Hirschl, and Ioi Lam as their hard work made possible the success of our project.

This work was supported by a grant from the National Science Foundation (IRI 94-10470).

8 References

[1] K. Arnold and J. Gosling. The Java Programming Language. *Addison-Wesley Publishing Company*, 1994.
[2] B. Bailey, J. Konstan, R. Cooley, and M. Dejong. Nsync – A Toolkit for Building Interactive Multimedia Presentations. *Proceedings ACM Multimedia*, 1998.
[3] S. Iyengar and J. Konstan. TclProp: A Data-Propagation Formula Manager for Tcl and Tk. *Proceedings of the 1995 Tcl/Tk Workshop.*
[4] http://java.sun.com/marketing/collateral/jmf.html.
[5] R. Johnson. Tcl and Java Integration. http://www.scriptics.com/java/tcljava.ps.
[6] I. Lam and B. Smith. Jacl: A Tcl Implementation in Java. *Proceedings of the 1997 Tcl/Tk Workshop.*
[7] J. Levy. A Tcl/Tk Netscape Plugin. *Proceedings of the 1996 Tcl/Tk Workshop.*
[8] M. McLennan. [incr tcl] – Object-oriented Programming in Tcl. *Proceedings of the 1993 Tcl/Tk Workshop.*
[9] J. Ousterhout. Tcl and the Tk Toolkit. *Addison-Wesley Publishing Company*, 1994.
[10] L. Rowe and B. Smith. A Continuous Media Player. *Network and Operating Systems Support for Digital Audio and Video. Third Int'l Workshop Proceedings*, 1992.
[1 1] A. Safonov. Extending Traces with OAT: an Object Attribute Trace package for Tcl/Tk. *Proceedings of the 1997 Tcl/Tk Workshop.*
[12] B. Smith, L. Rowe, and S. Yen. Tcl Distributed Programming. *Proceedings of the 1993 Tcl/Tk Workshop.*
[13] S. Stanton and K. Corey. Tcl/Java: Toward Portable Extensions. *Proceedings of the 1996 Tcl/Tk Workshop.*
[14] J. Swartz and B. Smith. A Resolution Independent Video Language. *Proceedings ACM Multimedia*, 1995.

Visualizing Personal Web Caches with Caubview

Charles L. Brooks
GTE Internetworking
clbrooks@bbn.com
Murray S. Mazer
Curl Corporation
mazer@curl.com
Frederick J. Hirsch
The Open Group Research Institute
f.hirsch@opengroup.org

Abstract

Caubview is a companion visualizer for the Caubweb system: Caubweb enables a user to create a local collection of Web documents to read and update when disconnected. Caubview allows the visual selection of alternative views of the cache as well as reorganizing and restructuring these views. This paper describes our ongoing work on the Caubweb system, focusing on its visualization component, Caubview. We describe the relationship between these two systems, how we developed Caubview by re-using code from the HistoryGraph application, our results, and our plans for further development. We further describe our experiences using Tcl/Tk as the development language and "programming culture" for these applications, and indicate how ongoing developments in Tcl/Tk have influenced our work. We conclude with some observations concerning our future use of Tcl/Tk, and recommendations for ongoing efforts for the Tcl/Tk community.

Introduction

We have been working with Tcl/Tk for over two years, building systems to simplify and improve a user's experience of the World Wide Web. We started with the HistoryGraph visualizer, a tool designed to automatically capture and display a user's browsing history in a tree, and allow manipulation and use of the representation. We then modified and reused this code to provide a visualizer for a more ambitious Tcl/Tk project, Caubweb. The Caubweb system is designed to provide disconnected Web access through the use of cached Web resources. The Caubweb system includes Caubview, the visualizer, and Cobweb, a library of reusable Tcl/Tk components.

Tcl/Tk initially gave us a rapid development environment as well as platform portability and extensibility. As the system became larger and more complex over time, Tcl/Tk also introduced some difficulties. A major issue has been the lack of compatibility from one Tcl/Tk implementation to the next (as an example, the TkNT4.0 supported both a `send` and a `DDE` command under Windows/NT: both these capabilities disappeared when we moved to Tcl7.6/Tk4.2). Initially we found the need to build custom Wish's for different extensions cumbersome, and we are pleased to now be able to use dynamic loading of libraries. As the system became larger, naming became an issue. Although the namespace facility in Tcl 8.0 solves this problem, we have a large body of older code to re-write. Writing object-oriented code has also been an issue. We used the *obTcl* extension because it was a pure Tcl implementation, but now [*incr Tcl*] seems to be the object system of choice (and *obTcl* is no longer actively supported). Converting to Tcl/Tk 8.0 was relatively painless, and the improvements are noticeable. Native look and feel helps meet our portability goals, and several other problems have been solved. At this time, it is difficult to decide in retrospect if we should have used Perl or some other choice instead of Tcl/Tk. The answer really will depend on what happens with the next release of Tcl/Tk, as well as ongoing work in the Java space with Jacl and Tcl Blend.

This rest of this paper discusses the development of the Caubview visualizer, how Tcl/Tk helped and where it raised issues, and how we addressed these issues by writing specialized Tcl/Tk code or using extensions. We conclude by summarizing some general issues with the Tcl/Tk "culture" (such as missing features, the release process, and a CPAN (Comprehensive Perl Archive Network [http://www.perl.com/CPAN/]) equivalent), and discuss alternatives to Tcl/Tk, such as Jacl and the Java Foundation Classes (Swing).

Background

Caubview began it's existence as a way to visualize and manipulate the resources contained in a Caubweb cache: it sprang to life as a quick re-write of the HistoryGraph [Hirsch97a] application, in combination with the Cobweb libraries. Since that time (May 1997), Caubview has become an important application in its own right, particularly in the viewing of shared caches amongst various cooperating Caubweb systems.

Caubweb and Caubview were written in Tcl/Tk from the beginning. Tracking the various Tcl/Tk releases over the life of this project has been both frustrating and rewarding. Choosing Tcl/Tk was not only a choice of a programming language, but a choice of a programming "culture", complete with local idioms, beliefs, attitudes, and styles. The story of Caubview and Caubweb can't be told without focusing on Tcl/Tk as well.

Caubweb

The Caubweb system is extensively described in [LoVerso97]. The following paragraph is taken from the introduction.

> Caubweb is a research system for investigating ways to provide adaptive, on-going read and update interaction with Web-based information, even under conditions of variable or intermittent network connectivity. Caubweb is part of ... [the] Distributed Clients project, which has the broad goal of increasing the availability and customization of Web-based information services for mobile computing users. The expected benefits include increasing the availability of information, reducing the latency of servicing requests, and adapting information to the specific user and context.

At base, Caubweb aims to provide a service more akin to a history-list mechanism than an actual cache (since, in fact, Caubweb violates HTTP/1.1 caching policy [Fielding96] in order to provide the user with a simulated experience regarding resources they have previously accessed). Architecturally, Caubweb is built as an HTTP proxy that uses a cache to meet its goal of providing access to Web resources when disconnected.

As part of our initial goals of platform portability and extensibility, the Caubweb interface design supported cache viewing via a "control panel", implemented as a set of dynamically generated HTML pages that enabled several alternate views of the cache. As part of this interface, the user could obtain a listing of HTTP servers for which resources were available, and, for each server, a list of all URLs retrieved from that server organized alphabetically, and with a additional information such as size. Locally modified documents are marked with a specific icon. Another view list all modified documents in the cache.

Our initial goal in creating the Caubview application was thus to both duplicate and improve on the original cache viewing mechanisms. We wanted to permit the visualization of various relationships among the resources in the cache and to determine the success of an automated retrieval process (that is, whether a particular weblet retrieval had fetched all documents of interest).

HistoryGraph

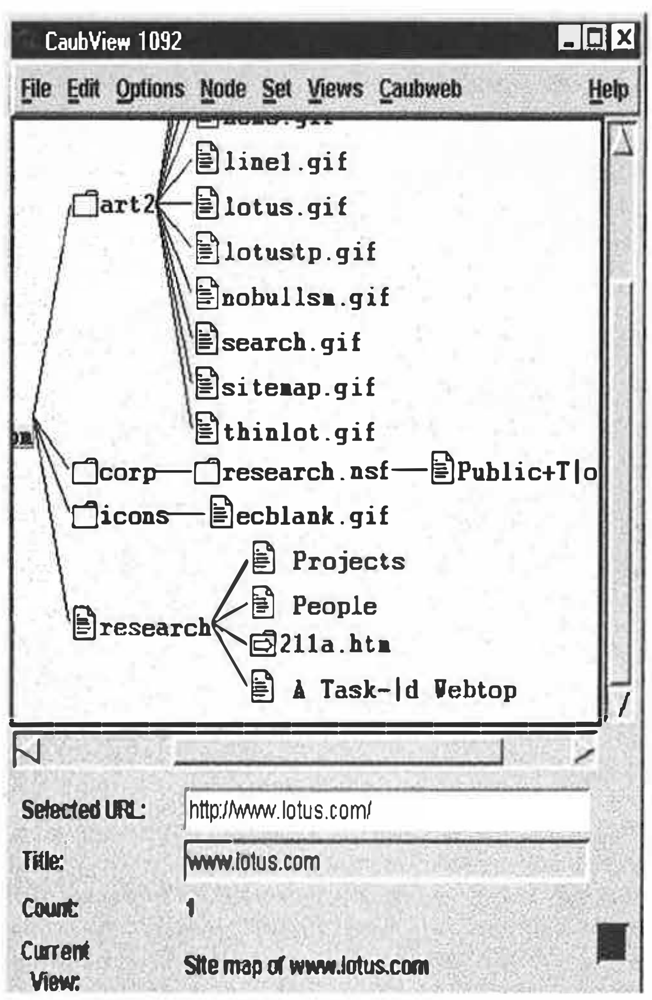
The HistoryGraph visualizer is a tool that provides a graphical history of Web browsing activity by creating a tree structure [Brighton97] that reflects the user's browsing activity. Our initial experiences in developing this application with Tcl/Tk are detailed in [Hirsch97b]. At base, HistoryGraph is a desktop browsing associate [Meeks95] that "automatically tracks user browsing activities, presents a graphic visualization of this activity, and provides a mechanism for manipulation and use of that history". The visualizer generates a tree of the user's browsing activity; where each node represents a visited URL and each arc indicates that the user has visited that URL via the URL represented by it's parent. The resulting view is not static: a user can graphically reorganize and prune the resulting tree in order to restructure this information, and the resulting tree can be saved across sessions and shared with others. Named sets of pages can be generated by selecting nodes from the tree and assigning them to a named collection; in turn, these sets can be saved for future use or forwarded to other applications for further processing and modification.

Caubview design

Caubview was derived from HistoryGraph and initially reused its design of a single display window. Development time for initial re-work was a little more than a week. Instead of receiving input from the user's browser and building the tree dynamically, Caubview reads the contents of a Caubweb cache index file on start-up, and generates an internal representation of the contents. The original goal for Caubview was thus to provide viewing of the cache when disconnected from the network. Initially, no attempt was made to synchronize access to the cache with Caubweb itself, nor were any attempts made to update the view of the cache once the initial index had been loaded.

Views

A Caubview visualization is organized around a series of "views": a *server* view, a *site* view, and a *closure* view. The initial view that the user sees is the "server" view, that shows a tree of the top level servers, organized alphabetically by host name. Each server view has a corresponding site view that is accessed by selecting a particular server. This view generates a tree of all resources in the cache that are provided by that server, organized by URL name. This is a *syntactic* arrangement, where leaf nodes (actual cache resources) are represented graphically with file icons(🗎), and intermediate path names are presented using directory icons (☐). An arc implies that the node is "lower" in the naming hierarchy than its parent, and also implies that that element is resident in the cache. Figure 1 below shows a site view of the cache.



Selected URL: http://www.lotus.com/

Title: www.lotus.com

Count: 1

Current View: Site map of www.lotus.com

Figure 1 Caubview display of a site view

In the *closure* view, a particular resource is selected, and new elements are generated, organized by embedded hyperlinks (we call this a *semantic* or a *link view*). In the semantic view, an arc indicates that the child node is the destination anchor for a hyperlink appearing in the parent node. Two interfaces are provided for generating closures: a "canned" series of closures, and a menu interface that provides more control over selection. A closure is defined by the depth in the tree to search, whether to include nodes only from the current server or all servers, and whether to display in the same tree of different trees: the interface allows these elements to be set separately. The results of a closure can be overlaid on an existing window or shown in a new window. The resulting tree can contain "ghosts": resources that are referenced by a hyperlink but are not present in the local cache. Ghosts are drawn using a different icon from a leaf node (⊡). Resources from a different server than that of the selected document that are also present in the cache are also shown with a distinctive icon.

Shared Caches

We recently have extended Caubweb to support the notion of shared caches, defined by a group of cooperating Caubweb servers. Caubweb itself was modified in the following way: if running in "cooperative" mode, and a resource is not found in the local cache, Caubweb would then query one of its defined peers (via an HTTP proxy request) to attempt to discover the resource. We initially use a simple list traversal algorithm instead of a more general flooding algorithm to determine if the resource is held by one of the cooperating peers. Looping is prevented by the addition of a Caubweb-specific header to the request prior to sending it to a cooperating peer: this prevents the peer from attempting to retrieve this resource from the origin server.

We also modified Caubview to support viewing of the shared caches. When running as a member of a shared cache group, the server view shows an aggregate view of all WWW servers available in all caches. Two icons are used to indicate either a local server (resources are available on the local machine) or a remote server (resources are only available remotely). Servers for which resources are stored both locally and remotely are marked as local. The site views are modified accordingly to show all resources, whether local or remote: in this view, remote resources are displayed with an icon reminiscent of a pointer, and the URL is used to name the resource instead of the document title (assuming it exists). Closure views are extended such that if the search depth indicates that resources should be analyzed for embedded links, the resource is fetched from the remote system and its link set is extracted from the HTML source. Referring back to Figure 1, the node labelled with a 🗗 icon (and named 211a.htm) is a reference to a remote resource.

Initial Results

Our initial experiences with the Caubview visualizer were mixed. We were disappointed by the initial reaction of people to whom we demonstrated the prototype. Most users responded positively: they were better able to understand the cache layout, and appreciated the restructuring capabilities of Caubview, since it retains most of the capabilities of HistoryGraph such as generating sets, manipulating the tree via drag-and-drop, and viewing resources via a specified browser (the browser is configured to proxy through Caubweb, which is running in off-line mode).

However, several users were confused between the semantic and syntactic views (especially when overlaid in the same window) as to whether or not the indicated resource was present in the cache. We believe that this was caused by differences in meaning of the arc relationship: in one case, an arc indicates a *naming* relationship (as well as existence); in another instance, it indicates a *linking* relationship (irrespective of cache residency). We are currently evaluating ways to better represent these relationships, possibly by using color or font changes to represent cache residence.

Ongoing Work

We are currently working on two issues: how to better visualize the cache hierarchies, and how to visualize metadata associated with individual resources and aggregates (servers, document collections [a set of resources meant to be viewed as a whole]), or metadata shared across elements). Underlying these investigations are the deeper issues of *navigation* versus *exploration*. For us, navigation implies both context and ways to generate various traversals of the information space based on existing hyperlinks. Exploration, on the other hand, involves the generation of new hypertexts (relationships, or the results of queries) by exploiting communality amongst resources.

Cache Visualization

When thinking about cache visualization, the first question that comes to mind is: How much of the hierarchy needs to be in view at any one time? Although our existing tree widget supporting panning of the display as well as scrolling, it is constrained by the demands of screen real estate, which is a constant tradeoff between showing multiple entities whilst still enabling the identification of those entities by either the last component of the URL path (the "file name", as it were) or by the compression of the title for the resource (providing one is given). One possible extension we have considered is to enhance the existing tree widget to support collapsing of individual subtrees. We have also evaluated using the Pad widget from Pad++ [Bederson96] or a hyperbolic display widget [Lamping95][Munzner95] as a way to represent more nodes and relationships within a given space.

We are also exploring linking to other programs (hypertools) via automatic methods, either via a publish/subscribe model of sets (where Caubview would define a selection of sets and their description, and other tools could subscribe to these sets), or via a more generic model, where certain hypertools announce their availability to provide certain services, and Caubview can then avail itself of these services. We have already experimented with the latter approach in the HistoryGraph application using the WhatsNew hypertool: this tool interrogates a list of resources and determines if they have (or have not) changed since a specified date and time, and provides a list of resources that meet that criteria. The response is returned to the HistoryGraph application by invoking a callback function, that results in a new named set being generated. Selecting this set via the menu bar will cause all resources in the tree to be highlighted. The user can then navigate through these resources and view the changed pages via their chosen browser. This capability is still latent in the Caubview application, but a more generic API needs to be defined.

Finally, given alternate ways of visualizing the local cache, we must then determine how the actions of pruning, hiding, and selecting would function in this new context. For example, given an infinite display space, pruning might be achieved by "unhooking" the subtree and moving it off-screen.

Metadata

A second focus in the next phase of our work is in the representation and visualization of metadata stored with Web resources This metadata can include information provided by a shared descriptive framework (such as that defined by the Dublin Core metadata framework [Dublin98]), or can be made available by specialized servers that provide organization-specific characterization of resources, such as relevancy to a particular task or role. One possibility is to use extra gestures (e.g. right click brings up metadata window, with information organized as a property list) as a means of directly viewing associated metadata. We are also evaluating the use of the Pad++ techniques of portals and "magic lenses" as alternatives. Using these techniques, a user would choose a selection criteria, and resources matching that criteria would be either highlighted in the display or transformed in some lens-specific fashion. All this activity is basically an application of the Visualization Mantra: *Overview, zoom and filter, then details on demand.* [Shneiderman97]

Successes

Once again, direct manipulation of a view (in our case, the cache hierarchy) has proven to be an effective and powerful UI technique. Unlike many Model/View based systems, manipulation of the *view* of the cache actually doesn't modify the actual contents of the cache: rather, what is modified are the *relationships* amongst the various cached resources. Thus, re-drawing of the tree via drag-and-drop becomes a way to provide "visual commentary" - creating (pseudo) hyperlinks (and hence personal associations) where none previously existed. The ability to save the modified tree for further manipulation or to define sets of resources that can be handed off to other applications is another powerful feature that supports re-use of existing hypertools, as well as supporting such search related tasks as history and progressive refinement [Shneiderman97].

While we agree on the need for better visualization tools to describe hypertext based information systems, we are still debating the issue of what works best in what situation. Are 3-dimensional representations more effective? 2.5 dimensional? Collapsible Trees (hierarchical) displays? We see our on-going challenge as answering the latter question in the context of our overall goal: namely, to provide our users with better information regarding the precision and relevance of their cached resources to their immediate tasks when disconnected from their network information servers. We believe that providing this context will consist of

- visualization (and direct manipulation) of hyperlinked information spaces;
- making metadata available as part of visualization, and
- the display of self-organizing (via metadata specifications) collections versus user-defined collections (based on some arbitrary criteria, whether that be content, metadata, or ad-hoc characterizations).

Tcl/Tk: Programming and Culture

Caubweb and Caubview have been based in Tcl/Tk since the beginning of the project. As such, we have gone from TkNT4.0 and Tcl7.5/Tk4.1 through the current Tcl/Tk 8.0 release. As is the case with several other programming languages (Perl and Python come to mind), Tcl/Tk is both a programming language and a programming culture, that provides its own sets of idioms, beliefs, and patterns of development. Our initial decision to move forward with Tcl/Tk was based not only on our desire for a rapid development environment, but also on our desire for platform portability and extensibility: our project requirements targeted Unix and Win32 systems (Windows/95 and Windows/NT) from the beginning.

Code Reuse

We were gratified by the amount of code reuse between the HistoryGraph and the Caubview application (move than 75%), as well as the ease of developing our initial implementation. We have had to re-write some of the code in porting from TkNT4.0 to Tk4.1 and thence to Tk8.0, specifically in the area of sockets and communication with other processes under Windows/NT (specifically Netscape Navigator and Internet Explorer). We have successfully used several other Tcl extensions, including Alan Brighton's Tree widget for display, and Tcl-DP [Perham97] for UDP and IP multicast sockets. In the former, we rebuilt the widget for Tcl8.0 ourselves; in the latter, we imported a binary distribution for Tcl8.0, and have since had occasion to rebuild the system on both Unix (Linux) and Windows/NT, largely without incident.

Both of the above extensions make good use of the `package` facility, although neither as yet utilizes the `namespace` facility as a means of preventing namespace collisions. We advocate strongly that any extension writer begin to use the `namespace` facility, especially when writing internal support routines. We have had to track down a couple of problems with duplicate definitions of `lremove` or `ldelete`, and while such bugs are not difficult to fix, they do take time from other more productive tasks.

Another important aspect of code reuse is the ability to integrate facilities from different applications in order to create new ones. We were able to easily replace home-grown, ad-hoc code in the HistoryGraph with code from the Cobweb library (Cobweb is the shared library portion of Caubweb), specifically code to parse URLs and HTML source. Problems arose, however, when we came to do the supporting shared caches. As long as Caubview had functioned as an *off-line* visualizer, the issue of cache modification was moot: the cache would effectively never change (modulo the modification of existing resources, which would have created a separate cache entry representing the original resource). By using Caubview as a visualizer when Caubweb itself is active, we require that Caubview be kept up to date regarding new entries in the cache. Testing cache index modification times, and re-reading and re-generating the cache is not an alternative, since Caubweb's internal index is updated each time a new resource is fetched from the network, and only written to disk when necessary. We anticipate adding a Observable/Observer implementation to Caubweb such that Caubview can register as an observer of the cache, and have Caubweb notify it when a new entry is made in the cache. This will require code changes such that Caubview instantiates a cache object if stand-alone mode, otherwise, it must register with the running Caubweb application. We have added a command line parameter to Caubweb to allow Caubview to be started along with Caubweb: Caubview in turn can be started with a switch that indicates it was started from Caubweb (once the `send` capability is available under Win32, we can then determine if there is an interpreter named "Caubweb" present in our environment).

Porting to Tcl/Tk 8.0

In January of 1998, we began a port of the existing Caubweb and Caubview applications to Tcl/Tk 8.0 (hereafter Tkl8.0). Both had previously been using Tcl7.6/Tk4.2. The port to Tkl 8.0 was relatively trouble-free. We re-wrote our *obTcl* library (a small, pure Tcl object-oriented extension) to remove the "::" separators used to indicate class methods and instead used a more Java-like syntax using "." as a separator. We also needed to modify the use of `upvar`-ed arrays: this was accomplished by generating a similar name in a separately defined *obTcl* namespace. The resulting code was highly portable between Tcl7.6 and Tcl8.0 (although the 7.6 implementation has not been exhaustively tested). Both the Caubweb and Caubview application seems faster, although we have not conducted extensive measurements.

We have welcomed the addition of several new features in Tcl8.0, including the support for binary data, and the newly legitimized `fcopy` (*nee* unsupported0) command for Caubweb. Caubview has seen the most improvement, however. The native look and feel support in Tk8.0 has made the user interface more consistent and palatable, and the new font specification mechanisms has made font handling more consistent and the results less "surprising".

Tcl/Tk futures

In the time that we have been using Tcl/Tk for the Distributed Clients project, we have observed several changes in Tcl/Tk that has made the language more suitable for use in a larger projects. Chief amongst these changes has been the increased and better use of the `package` and `namespace` features added to the language in Tcl7.6 and Tkl 8.0. Prior to these features, one had to resort to a "surfeit of Wishes" via a series of statically linked extensions. Today, most extension writers are using dynamically loadable packages, and increased use of the `package provide` and the `namespace` facilities to provide better separation of function as well as better distribution of executable code (Don Libe's CGI package is an example). We are modifying Caubview and Caubweb to use these features more extensively as we re-work various parts of the system.

While Tkl 8.0 has been an major improvement for us, we are still waiting for functionality that was available almost two years ago in TkNT. We eagerly await the release of Tkl8.1 with support for the *send* command under Win32, as well as DDE support. Another feature we would dearly welcome is a working version of the capability to embed another applications window inside of Tcl/Tk under Win32, since this would provide us with another mechanism with which to integrate Caubview with Caubweb.

Jacl and TclBlend

As part of our ongoing visualization work with Caubview, we looked at hyperbolic graph widgets, and found at least two written in Java. Our problem was how to integrate Java widgets into our Tcl/Tk application, and with the 1.0 release of Jacl and TclBlend in early 1998, an answer seemed imminent. TclBlend supports the inclusion of Java classes in a Tcl program [Johnson97], while Jacl is a Tcl interpreter written in Java [Lam97]. However, Jacl 1.0 doesn't support Tk at all, and portions of Tcl (primarily dealing with asynchronous file I/O) are missing as well. In addition (as of April, 1998), the TclBlend extension is advertised not to work with the upcoming Tkl 8.1 release (which we need for the `send` command and DDE support for Win32), and also requires a native thread implementation of Java, that constrains our choice of target systems. This means that if we are to continue to leverage our investment in the Cobweb library, as well as gain the new functionality we want in the 8.1 release, we are forced to abandon Java integration with Tcl Blend; if we choose Jacl, we will have to re-implement certain portions of the library that requires Tcl functionality not present in the 1.0 release.

The recent 1.0 release of the Java Foundation Classes (*aka* Swing) further compounds our dilemma, since Swing provides a number of useful Java widgets for building applications. Our current plan is thus to reimplement Caubview using Jacl, using Tcl Blend as an intermediate step. We will re-write the UI portions to use JFC, and work on integrating as much of the Tcl functionality as possible. In this way, we hope to (yet again) leverage our investment in generating platform portable and extensible code.

Back to the Future

We do not regret our decision to use Tcl/Tk, but we are not sure we would make the same decision again, especially with the advance of Java in the last two years. The problems we have encountered with Tcl/Tk over the years can be summed up as follows:

- Ongoing problems with integration of new extensions (installation scripts, namespace collisions). In addition, interfaces to various components are all just a little bit different (a PAD widget is almost a Tcl canvas widget, but not quite). There are also performance differences (sometimes vast) between extensions under Win32 and Unix.
- Each release of Tcl breaks some amount of working code
- Feature lag between platforms (e.g. `send` under Win32), and timely availability of older extensions.
- No coherent O-O strategy (*incr Tcl* seems to be the O-O package of choice, but, again, it lags the field).

Finally, it is still much too hard to determine what extensions are available, or to ascertain the quality of these extensions. It is our fervent wish that the Tcl community can create an organizational equivalent to CPAN (perhaps ETEN, the Exhaustive Tcl Extension Network)).

For us, cross-platform functionality *and* equivalent performance are critical for the ongoing success of Tcl/Tk, and we urge developers of Tcl extensions to provide support for both Unix and Win32 platforms. There is increasing evidence that Windows/NT will continue to gain importance even within the research community. Our own work environment combines Win32 desktop machines linked to Unix file and compute servers, and we expect this situation to become even more common.

Given the above, how should Tcl/Tk development proceed in the immediate term? We believe that in order to be successful, Tcl needs to return to its roots: a small, easy-to-use embedded interpreted language, acting as a "glue language" for scripting other components. Just as Tcl/Tk provided this capability for the combination for C and the X graphical widgets, we believe that Jacl can provide this capability for Java and the AWT/JFC widget set. The combination of Tcl scripting and Java components thus holds great promise for stemming the tide of Wintel hegemony, and providing a powerful cross-platform solution for application developers.

Acknowledgments

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under the contract number F19628-95-C-0042. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, or the Defense Advanced Research Projects Agency or the U.S. Government. The research efforts upon which these findings are based was performed at the Open Group Research Institute.

Special thanks are due to John LoVerso and Scott Meeks.

Caubweb is a trademark of The Open Group Research Institute. Other trademarks are the property of their respective companies.

Availability

The Distributed Clients project has completed. The last release of Caubweb is available at http://www.camb.opengroup.org/RI/secweb/dis_clients/oasis.html. Caubview remains a moving target: those interested in more information should contact the authors.

References

[Bederson96] Bederson, B., Hollon, J. D., et. al. *Pad++: A Zoomable Graphical Sketchpad For Exploring Alternate Interface Physics.* http: //www.cs.unm.edu/pad++/papers/jvlc-96-pad/index.html

[Bederson97] Bederson, B., Hallon, J., "A Zooming Web Browser", *Proceedings of SPIE Multimedia Computing and Networking* 1996, Volume 2667, pp 260-271 ftp://ftp.cs.unm.edu/ pub/pad++/spie96_html.ps.gz

[Brighton97] Brighton, A., *Tree-4.0.1 - A Tree Widget for Tk4.0 based on C++ and [incr Tcl],* http://www.neosoft.com/tcl/ftparchive/sorted/devel/tree-4.2.README

[Dublin98] *Dublin Core Metadata,* http://purl.oclc.org/metadata/dublin_core/, valid as of 4/6/98.

[Fielding96] Fielding, R., et.al, "Hypertext Transfer Protocol: HTTP/1.1", *World Wide Web Journal,* 1(4), Autumn, 1996, O'Reilly and Associates, pp. 89-186.

[Hirsch97a] Hirsch, F.J., Meeks, W.S., & Brooks, C., "Creating Custom Graphical Web Views Based on User Browsing History", *Poster Proceedings, 6th International World Wide Web Conference,* http://www.opengroup.org/www/waiba/papers/www6/hg.html

[Hirsch97b] Hirsch, F.J., "Building a Graphical Web History using Tcl/Tk", *Proceedings of the 5th Tcl/Tk Workshop,* USENIX Association, pp. 159-160.

[Johnson97] Johnson, R., *Tcl and Java Integration,* http://www.sunscript.com/java/tcljava.ps (as of 3/30/98).

[Lam97] Lam, Ioi K., Smith, Brian C., "Jacl: A Tcl Implementation in Java", *Proceedings of the 5th Tcl/Tk Workshop,* USENIX Association, pp. 31-36.

[LoVerso97] LoVerso, J., & Mazer, M.S., "Caubweb: Detaching the Web with Tcl", *Proceedings of the 5th Tcl/Tk Workshop,* USENIX Association, pp. 19-29.

[Lamping95] Lamping, J., Rao, R., Pirolli, P. (1995) "A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies", *Proceedings Chi'95,* ACM SIGCHI, New York.

[Meeks95] Meeks, W.S., Brooks, C.L., Mazer, M.S. "An Architecture for Supporting Quasi-agent Entities in the WWW", *Intelligent Agents Workshop Proceedings,* ACM Conference on Information and Knowledge Management, December 1995, http://www.opengroup.org/ww/waiba/papers/CIKM/CIKM.htm

[Munzner95] Munzner, T., Burchard, P., (1995) "Visualizing the Structure of the World Wide Web in 3D Hyperbolic Space", *Proceedings of the First Annual Symposium on the VRML Modeling Language,* ACM SIGGRAPH, New York, pp. 33-38.

[Perham97] Perham, Mike, et. al., "Redesigning TCL-DP," *Proceedings of the 5th Tcl/Tk Workshop,* USENIX Association, pp. 49-54.

[Ousterhout93] Ousterhout, J. and Rowe, L.A, "Hypertools: A GUI Revolution," *The X Journal,* 2(4), March-April 1993, pp. 74-81.

[Shneiderman97] Shneiderman, B., *Designing The User Interface,* 3rd edition, Addison-Wesley, 1997, p. 523.

Using Content-Derived Names for Package Management in Tcl

Ethan L. Miller & Kennedy Akala
Computer Science & Electrical Engineering Department
University of Maryland Baltimore County
1000 Hilltop Circle
Baltimore, MD 21250
{elm,kakala1}@csee.umbc.edu

Jeffrey K. Hollingsworth
Computer Science Department
University of Maryland
College Park, MD 20742
hollings@cs.umd.edu

ABSTRACT

Managing different versions of library routines has long been a problem, both for Tcl and for other languages that permit code reuse and modification (i.e., all computer languages that the authors are aware of). This problem is particularly difficult for Tcl because it allows libraries (in the form of packages) to be dynamically loaded as needed. While this feature is very convenient — users need only keep a single copy of each library to use it in many programs — it can lead to code compatibility and distribution problems.

This paper presents a solution for this problem — using content-derived names (CDNs) to name Tcl packages. Using this solution, a program can simultaneously use two different versions of a single package. In addition, the Tcl interpreter can easily find instances of a missing package over the network and download them, making them available to a running application. Because content-derived names are computed using a cryptographically strong hash over the text of a package, this process is safe from spoofing and other attacks based on providing the wrong library. Thus, a user may download missing packages from any server willing to provide them without fear of virus or trojan horse attacks.

1 Introduction

The proliferation of complex software libraries has made development easier in Tcl as well as in other languages by providing high-level functionality to application programmers. However, these libraries also complicate matters by introducing potential incompatibilities between an application and the packages that it wants to use. While Tcl can use the `package` statement to deal with this problem, there are still many shortcomings that need to be addressed. This paper discusses an extension to traditional Tcl packages that eases the distribution of large Tcl applications and allows the inclusion of two different versions of the same library, as may be required by large applications that include packages that themselves require packages. Additionally, the `tcdn` (Tcl Content-Derived Name) mechanism permits applications to automatically download needed packages over the network, even from untrusted hosts, and insert them into running code. Using this package, application distribution changes from a `tar` file with dozens of Tcl files and a `README` list of required packages to a single "root" Tcl file. All of the remaining code can be dynamically fetched as needed.

In additional to making distribution of applications much easier, `tcdn` permits many versions of packages to coexist peacefully on a single machine. The Tcl `package` mechanism currently allows this, but only supports "exact" or "later than" testing for package version numbers. In our experience, however, the "later than" approach is dangerous; often, changes in a library from version 2.x to 2.(x+1) break some applications that used the earlier version. In such a case, the application designer is deluged with requests to squash bugs. Using the `tcdn` system, however, a developer can supply an application and specify *exactly* which packages and files must be used with it. Since the developer controls the environment more precisely, she is better able to test the application's behavior.

The `tcdn` package provides a third benefit: security. Content-derived names are computed by hashing the contents of a Tcl package using a secure hash such as MD5 [6] or SHA-1 [1] and recording the (relatively short) result in the file that uses the package. It is a simple matter for a Tcl file to recompute the hash value before importing a file, guaranteeing that the application is indeed using the appropriate file.

2 Background

While the idea of using content-derived names (CDNs) for configuration management of executable code is new, there has been previous work in the areas of using explicitly managed version numbers to provide configuration management. The `tcdn` package builds on this work as well as research in secure hash functions.

2.1 Configuration Management

Most of the research in configuration management has concentrated on managing the construction of applications from a source code repository. This approach can work well with object code binaries, producing a single monolithic executable object that may be distributed. If this is the case, there is no need for further management of multiple versions of the same code. While the software developer must keep track of many versions of code, the end user need not. As a result, much commercial software is distributed this way.

Increasingly, however, software developers are providing their applications as a collection of code objects. The use of dynamically linked libraries in Unix, MacOS, and Windows facilitates programs' use of standard code. With script-based languages such as Tcl, though, distribution of applications as dozens or hundreds of individual files is practically guaranteed. Managing these files can cause big problems, as the authors have experienced when installing programs on their personal computers. Each application provides the libraries that it needs in the versions that it prefers, overwriting any existing versions of the libraries. Of course, this approach causes some existing programs to fail because their preferred version of the library has been erased by a later installation. Tcl provides a mechanism to avoid this, but its use requires detailed knowledge of package search paths. Additionally, file names can become a problem because different versions usually use the same file names, and package search paths can become exceedingly long. van der Hoek, et. al. [3] addressed this problem of "software release management" by suggesting a system to support software acquisition by ensuring that the correct versions of dependent packages are acquired with the primary package. However, their approach relies on a centralized software repository and explicit administration of version numbers for all packages. In contrast, our approach is completely decentralized and allows anyone to install a new application by simply entering a short (less than 50 characters) string of hexadecimal digits and a location from which to retrieve the object.

2.2 Package Versions in Tcl

Tcl has a mechanism to accommodate packages with different version numbers that permits applications to request a package with a specific version number or any version number "later than" a specific version. A piece of code may request a package using a `package require` statement; this causes Tcl to search through the package index file for an entry that provides the package. This index is built by searching files for `package provide` statements. The index file is constructed by looking through files in the order specified by a Tcl specific variable, and it must be constructed statically (though it could be run automatically when an unknown function is encountered). Nonetheless, the standard Tcl approach requires that a user install all required packages before running an application.

While this approach can work with "well-behaved" packages, it presents several difficulties. First, users must make sure that all files are available before running the application. While attendees at the Tcl/Tk conference may have no difficulty doing this, average users will have somewhat more trouble. Package availability is only the start, though. Another issue is version management. Use of versions greater than the one with which the application was tested can cause bugs in a program. While developers would like to think that version 2.2 is fully backward compatible with version 2.1, this is often not the case.

To address this problem, developers who care about their code working should use the `exact` option to the `package require` statement, allowing them to test their code with all of the files that it will use. This approach introduces another problem, however. With complex code, it is possible that a single application may require two versions of a single package. The high-level code may not even be aware of this conflict if two packages themselves each require a different version of the same lower-level package, as shown in Figure 1. In Tcl, this conflict cannot be easily resolved because only one of the `package require` statements will be able to load the desired package. The developer of "root object" may not even know of the conflict if she received the code for the two top-level packages from different sources. This can also introduce naming problems for unwary code designers because it requires that every version of a package have a unique file name. Of course, this can be done by appending the version number to every file in a package, leading to the problem of deleting old files when the package using them is gone.

2.3 Secure Hash Functions

A key feature of `tcdn` is the use of a secure hash function to assign a unique name to an object based solely on its content. Digital signature algorithms such as MD5 [6] and SHA-1 [1] are one-way functions that take arbitrary data and produce a result that is very likely to be different from that of any other (different) input sequence. Our implementation uses MD5 to generate CDNs, but other algorithms could easily be substituted.

Because it is NP-hard to find another object that produces the same digital signature as a given object, it is

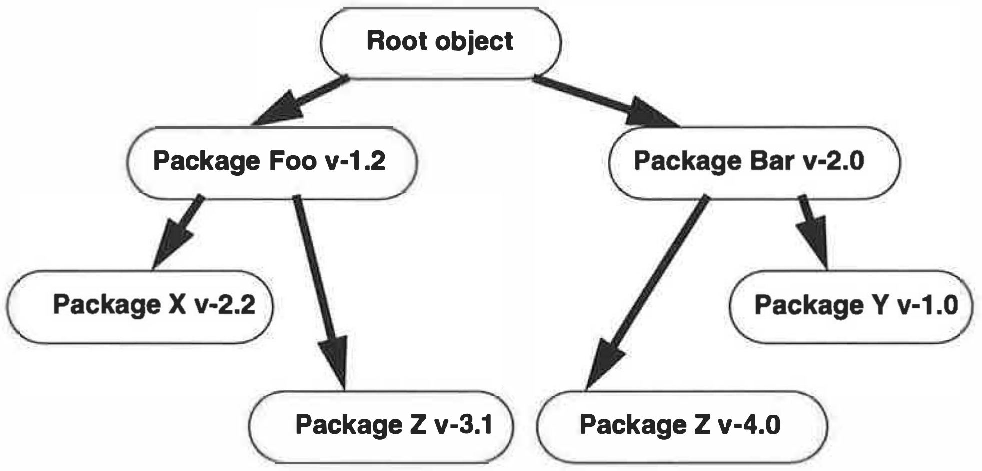


Figure 1. Package requirement conflicts in a complex application.

unlikely that two objects will have the same signature, either by chance or by malicious construction of an object. For the 128-bit signature provided by MD5, the chance of two objects out of 10^{15} having the same signature is approximately 10^{-9}. By increasing the signature length to 256 bits, the chance of collision drops to 10^{-14} for 10^{30} unique objects [4].

The use of secure hash functions provides another benefit beyond conflict-free naming, however. It allows applications to ensure that the code they are loading is authentic, preventing the introduction of trojan horses. This concept is also discussed in [5]. If a developer has a virus-free environment (and we hope that they do), the hash values that they compute will be those for correctly working code. If a virus later infects any piece of code, the secure hash will change and the loader will be able to reject the package, instead choosing to download a new version from the network.

3 `Tcdn` Design

The basic concept underlying `tcdn` is that a complex software installation can be thought of as a directed graph of procedure calls, and that procedures are grouped together into Tcl packages. The user does not care about internal package names; names are for the convenience of developers only. While external function names are important to those using a package, the name of the package itself is still largely irrelevant — just a word to be typed into a `package require` statement.

Tcdn provides all of its benefits by converting package names from a name and version number meaningful to a developer into a content-derived name that ca n be used to check package integrity and support secure remote retrieval. Since this name is probabilistically guaranteed not to conflict with other package names, it may be shared between different computers without fear of name duplication.

3.1 High-Level Design

The goal of tcdn is to convert a directed package graph such as that in Figure 1 into a package graph using content-derived names, such as that shown in Figure 2. Unlike the graph in Figure 1, which might have two `z.tcl` files (one for version 3.1 and one for 4.0), the graph in Figure 2 has unique names for all packages. Moreover, the code for `foo-1.2` includes `tcdn-package require` statements that reference the packages that it uses — in this case,
`1ee91c2024d8dbe901a33bf3b3200afe`
and
`42faca939af96f68ac164858cffdbc96`.
Because the content-derived name for `foo-1.2` is a cryptographic hash over its entire code, including the statements that reference the packages used by `foo-1.2`, it is impossible for a malicious user to change the references to the two packages without changing the hash, and thus the CDN, of `foo-1.2`.

More generally, the user need only trust a single Tcl package, that which contains the routine that is called to start the whole application. If the name for that object is obtained from a trusted source (perhaps as part of a financial transaction in which a user purchases the software), the user can obtain the root object itself, as well as all objects it requires, from any computer willing to provide them. If the user does not trust other servers (a wise precaution today), she can check the cryptographic

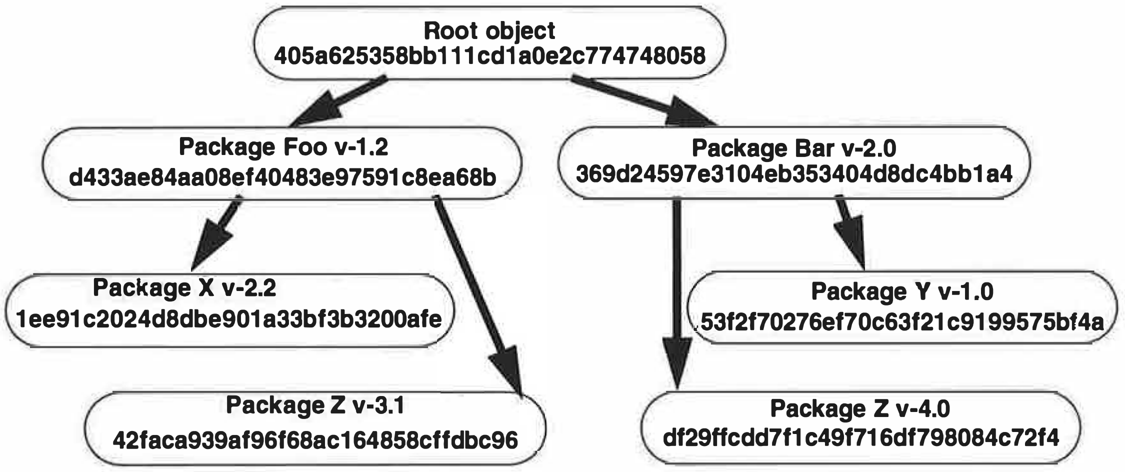

Root object
405a625358bb111cd1a0e2c774748058
Package Foo v-1.2
d433ae84aa08ef40483e97591c8ea68b
Package Bar v-2.0
369d24597e3104eb353404d8dc4bb1a4
Package X v-2.2
1ee91c2024d8dbe901a33bf3b3200afe
Package Y v-1.0
53f2f70276ef70c63f21c9199575bf4a
Package Z v-3.1
42faca939af96f68ac164858cffdbc96
Package Z v-4.0
df29ffcdd7f1c49f716df798084c72f4

Figure 2. Packages converted to `tcdn` format. Note that each package has a unique 32 character name.

hash of a downloaded object against the name she provided. If they do not match, the object is faulty.

3.2 What the Developer Sees

A developer need not radically change the way she writes code to benefit from tcdn. Instead, she needs to follow a few simple rules. First, each package must have its own namespace. This namespace must be named so that its name is different from that of every other package, including different versions of this package. If two version of a package share the same namespace, they cannot each use different versions of underlying CDN-identified packages in a single program. Giving different versions of a package each a unique namespace is not difficult, however, because the version information can be appended to the namespace name to guarantee a unique name.

The second restriction on developers is that packages may not use mutual recursion. In other words, if package A requires package B, package B may not in turn require package A. The simplest way around this problem is to break up one of the packages into two pieces, removing the cycle in the package graph. An alternative solution would be to combine packages A and B into a single, larger package.

If the programmer follows the above guidelines, she may use the tools described in Section 4.1 to convert her code into tcdn packages, making them available over the Web.

3.3 What the User Sees

The user's view of a large Tcl application is greatly simplified using `tcdn`. Rather than having to download and place dozens of files, some of which may overwrite previous files, he simply requests a single object that automatically fetches other objects over the Web. There is no longer a need to add a new directory to the package search path for the new application, and users who prefer the old package may continue to use it with no naming conflicts.

4 Tcdn Details

The `tcdn` package includes two pieces: code run by an application developer to generate the content-derived names and rewrite packages, and runtime functions necessary to locate and load packages named by content.

4.1 Developing Code for `tcdn`

Packages to be turned into tcdn packages are in largely the same way as "normal" packages. There are, however, a few restrictions that must be followed to allow CDNs to work. The restrictions are:

- All `package require` statements must be placed in the appropriate namespace.
- Each package must be contained in a single file. As a result, each file must have a `package provide` statement.
- There cannot be any circular dependencies between packages.
- All package code should be enclosed in a namespace with a unique name. This can be done by appending the version number to the "original" namespace name. If this is not done, most of the `tcdn` functionality will still be available, including the ability to fetch missing packages from a remote server. However, a single program will not be able to simultaneously use different versions of a particular package.

Once the code is complete, a tool is used to rewrite all of the package names into content-derived names. This is accomplished using a Tcl procedure with similar semantics to `pkg_mkIndex`. The routine to perform name conversion is called as follows:

```
tcdn::tcdnify <destination> <source files...>
```

This call operates on all of the source files named in the command, and places the resulting CDN files into the directory named by `<destination>`.

After converting files with `tcdnify`, packages may then be distributed to other developers who can use the package with `tcdn::tcdnpackage` require or to end users. Of course, this distribution may include all of the files if desired, and this option is necessary if the destination will not have Web access. A much more attractive option, however, is to distribute the package by simply providing the content-derived name (the entire object can be sent, but is not necessary) to the user. Future invocations will then automatically fetch the desired objects from either your Web server or any other Web server that has a copy of the file. The user is assured of receiving the correct file because her computer can compute an MD5 hash over the downloaded file; only if the file matches is it used.

4.2 The `tcdnify` Process

Packages in `tcdn` are named using a secure hash run over the entire body of the package. This name is then embedded into all files that require the package.

The `tcdnify` procedure has three steps. First, it creates a list of packages, resolving any source statements it finds. Next, it orders the packages by their dependencies on each other. If package A requires package B, then package B must be converted first because the secure hash for package A depends on the content-derived name for package B. A sample dependency graph for the packages listed in Figures 1 and 2 is shown in Figure 3. The order in which files are processed is noted next to each file. Note that, in all cases, a package is processed after all of its children have been processed.

Once the files have been ordered, `tcdnify` runs through a loop for each package in order. For each package, all package require statements are converted to tcdnpackage require statements with the appropriate CDNs, and then the entire file is hashed with MD5. The result is stored in the specified destination directory.

Perhaps the most difficult part of this process is ordering the packages by their dependencies. While this could have been left out by simply requiring the user to convert a single package at a time, we felt that it was important to make the process as automatic as possible. As a

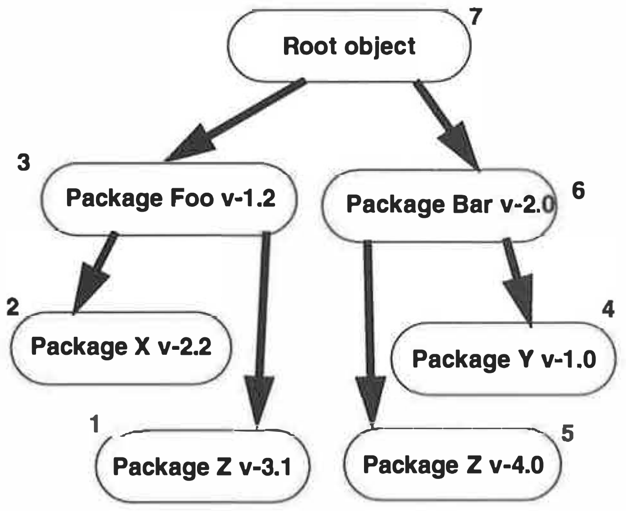


Figure 3. Sample package dependency graph.

result, a developer need only run `tcdnify` on an entire project to prepare it for distribution. Once this has been done, the resulting files can be made available for distribution via `http` or `ftp`, with only the root object distributed to potential users.

4.3 File System Independence

By assigning content-derived names, we guarantee that each version of each package has a unique name. Thus, we can store all packages in a single directory with no fear of name conflicts. Of course, the efficiency of a file system may drop when handling directories with potentially thousands of files, but this problem has been solved in the SGI XFS file system [7] and elsewhere. This arrangement eliminates the need for users to specify information about where the software packages will reside, and makes it simpler for a designer to test the software because she no longer has to worry about users with different package search paths. The authors have had difficulties with Tcl software that refuses to work until the ordering of a search path is changed; we believe that this approach to software distribution is flawed because it limits usage to those who are relatively good at software installation.

Because all package files reside in a single directory, the application will work regardless of what that directory is actually named. There is no need to embed directory information directly or indirectly into programs; instead, the `tcdn` system has a single directory (or list of directories, if desired) in which it looks for components. This directory (list) is stored in `tcdn::tcdndirs`, but it is relatively immune to user error. Should the user specify an incorrect (but readable and writable) directory, `tcdn` will merely download "missing" packages, even if they are stored elsewhere in the local system.

Unfortunately, this approach does not work for software developers who need to be able to modify packages because it forces name changes when the contents of a package change. However, it *can* be used for developers who wish to use other packages unchanged, and works very well for the majority of users who simply want their application to work without the need for painful installation.

4.4 Locating Packages

Perhaps the best feature of the `tcdn` package is that it permits automatic downloading of missing packages. If a required package is not found in the single directory that holds CDN-named packages, it may be fetched from a remote Web server using `http`. This can be done without the user's knowledge; the only evidence that the network was consulted is the increased delay.

When tcdn attempts to load a package, it first looks in the directories specified by tcdndirs. If the package is not found there, it proceeds to query each of the URLs contained in the list variable `tcdn::tcdnservers`. This list is searched in order, so it is likely that a site may put its own package cache server first before the "home" site or more comprehensive, but more distant caches. Additionally, the application itself can append values to `tcdnservers`, enabling an application to specify a Web server from which its component packages may be obtained.

By using this two-level approach, a site may maintain a cache of Tcl packages for use by many machines at the site. If the object is not available there, tcdn can go to either a public server with many packages (the equivalent of `sunsite`, perhaps) or to the developer's site to download the object directly from its source.

The integrity of packages found on the Web is of utmost concern because it is far too easy to implement a Tcl trojan horse. Thus, tcdn checks the integrity of any downloaded package (and, optionally, *any* `tcdn` package including those found locally) by hashing it and comparing its hash to its name. Files that do not hash properly are simply discarded, though it would be a simple extension to add Tcl code to send mail to a system administrator noting that a "bad `tcdn` package file" was received, alerting her of potential dangers. Note that if an integrity check fails, the package is treated as if it were never there. Thus, tcdn can go on to other servers listed in `tcdnservers` and check them to find a good package. If no valid package is found locally or on any server, `tcdn` throws an error.

Another advantage of this scheme is that it is not strictly necessary to even be able to store the file in order to use it. Instead, Tcl can dynamically load in the downloaded file but never store it on disk. This approach is poor for machines with disks that can cache the file locally. However, it may be advantageous for Tcl interpreters with no persistent local storage, such as those that run inside a Web browser.

5 Using TCDN

This section describes the actual installation and usage of the `tcdn` package. Because the package is simple, and places relatively few limitations on its use, it should be straightforward to use it with existing code. However, it will work best if developers follow a few simple guidelines for writing packages.

5.1 Installing `tcdn`

The `tcdn` package was written so that it can coexist and work with the existing package system. The `tcdn` package links the two worlds, and is both a `tcdn` package and a regular package. In fact, it is necessary to use the regular package mechanism to install and use the `tcdn` package. Usually, a system wide installation will entail placing the package in directories where writing is restricted, much as with any other normal package. This is only required for the initial installation of `tcdn`. Since the `tcdn` package is itself a CDN-based package, all later updates can be made automatically. Any package or application that uses the `tcdn` package needs to include a `package require tcdn` command. Following this, the package or application can use an upgraded `tcdn` package if it is available by including a `tcdnpackage require` statement. The semantics of using a CDN-named package will be covered shortly. For now, the important thing to know is that the original version of the `tcdn` package can be replaced at run time by any newer version if one is available.

The initial installation will also require a small amount of setup. The most important (and so far only) step of this setup is deciding where the downloaded CDN-named packages will reside. The surprising answer here is that the CDN-named packages should be stored in a completely public directory, readable and writable by all. Usually, this would be a problem because it would open the end user up to all manner of trojan horse attacks. If `tcdn` were not in use and packages were stored in this manner then any user would be able to replace a package with whatever they wanted. This would be like making `/bin` world writable! `Tcdn` protects against this by making sure that the package an application is loading really is the right one. With tcdn, the offending package would simply be deleted and replaced with the correct one. The details of this process are covered in the next two sections.

Keeping packages in a public directory is an immense advantage because it allows the end user to use an application without having to get the access required to download and install all of the packages required by the application. It also allows the installation process to be more completely automated, thus making distribution much easier.

5.2 Creating a CDN-named Package

The tcdn library was designed so that it would be easy for new CDN-named packages to be created. The goal here is to make things simple so that the programmer will not have to go to unreasonable lengths to create a CDN-named package.

There are several conventions that must be followed when creating a package. CDN-named packages should contain a single package provide statement. The name of the package does not matter because it will be removed. This is necessary to allow for the conversion of multiple packages with multiple files each at the same time. Global variables should, of course, be avoided. Namespaces are not required, but they are recommended. If no namespaces are used, tcdn cannot ensure that the correct version of required packages are loaded for the same reason that vanilla Tcl cannot do so. Care must be taken when naming namespaces, as namespace collisions can still occur. The easiest method of assuring a unique namespace is to append the version number of the package to its name and use that as the namespace name. Following this short list of rules should be easy, as it allows the programmer to create packages in a more normal fashion.

CDN-named packages should also contain a command named tcdnInit. This command should exist in the global namespace. The purpose of this command is to allow the package to do initialization if it needs to. Remember, a package may not have existed on a system prior to the first time it is used. The tcdnInit command will allow the package to perform any setup that it needs to. If the package does not need to perform any special setup then the command can be left out. The command is executed right after the package is loaded and before control returns to the application. It is very important that the command be as unobtrusive as possible, because it will be running in the context of the application.

CDN-named packages should store their configuration information in the user's directory. This is just like storing user options, and in fact just adds global options. The tcdnInit command should check for this configuration information before creating it or asking for it. The package programmer should make sure that this information will not take up too much space, and should also insure that any errors in its creation or reading will be handled without crashing the application. This requirement is not unique to tcdn, since no one would want to use a regular package that caused applications to crash.

Using another regular package in a CDN-named package is simple. Here the programmer should just use package require as usual. Of course, the package being requested must exist on the system where the package is used or the package require command will fail. Using a package in this way may be necessary sometimes, especially since not all packages may be available as CDN-named packages. If CDN-named packages were used all around then the normal package mechanism could be replaced completely, but until then CDN-named packages may require a regular package every now and then.

Of course, using a CDN-named package is just as simple. Tcdn provides a tcdnpackage require statement that handles the loading of CDN-named packages. In this case the requested package does not need to exist on the system at all, because tcdn will find and download it when it is needed. This frees the programmer from having to specify what packages are needed in order to be able to use their package. With tcdn the end user does not have to manually download the needed packages or even install them.

Once written, a CDN-named package must be converted. Tcdn was written as a library, so anyone can create an application that does the actual "conversion." This was done so that the process could be as flexible as possible. Tcdn provides a tcdnify command to do this conversion. The tcdnify command works on properly written regular packages that are to be converted to CDN-named packages. It strips the package of package provide statements and resolves any package interdependencies which may exist between packages being converted at the same time. It then outputs the CDN-named package with the correct content-derived name as the file name. The current version of tcdn does this because it is assumed that programmers will be more comfortable with creating packages in the manner they have been used to. Future versions will provide a mechanism for simply generating the name of a package that has been written as a CDN-named package from the beginning.

5.3 Distributing CDN-named Code

One of the primary design goals of `tcdn` was to make distribution much easier. The current package mechanism requires users to manually find and install packages as needed. `Tcdn` will automatically find and install CDN-named packages on demand. `Tcdn` can be configured with the location of CDN-named package servers. When a CDN-named package is requested and not found on the local system these servers are searched in turn. The package is then downloaded from the server and installed. This means that in order to distribute a package the programmer needs to upload it to a server or several servers. The programmer must also make public the content-derived name of the package. This name is all anyone else needs to know in order to be able to use the package. Application programmers do not even need to know the names of the servers on which the package has been stored. The end user doesn't need to know anything at all. Once the package has been uploaded and the content-derived name has been publicized, the entire process is automatic.

5.4 Using a CDN-named Package

Using a cdn package is easy. Because `tcdn` is not the primary package mechanism the application will need to have a `package require tcdn` command. This will load `tcdn` and all of its commands. From here all that is needed is a `cdnpackage require` statement for each CDN-named package that will be used. After this the process is automatic.

If the CDN-named package is located on the system it is checked and then loaded. The check involves regenerating the content-derived name. The generated name is then compared with the requested name. If the two match then the package has been located and verified and can be loaded. If the two do not match then it is assumed that the package file is corrupt and it is thrown away. If there are other directories to search then this process is repeated for each of them. If not, the package must be retrieved from a server.

The loading process for remote CDN-named packages is similar to the loading process for local packages. Each server is queried in turn for the desired package. `Tcdn` has been written so that different protocols can be used for each server. If none of the available servers returns the desired cdn package then the `cdnpackage require` command fails. This is not a normal situation, and would only happen if the network was unavailable or some other occurrence somehow prevented access. Usually, at least one server will return the requested package. The content-derived name is then verified, just as it would be if it were local. If the generated CDN matches the requested CDN then the package file is usable and can be saved. If not, the package is discarded and the process continues.

6 Future Directions

Having demonstrated the usefulness of CDNs in Tcl, we hope to extend our work to other languages. In particular, we plan to build similar functionality into the dynamic library loaders for Windows and Linux, allowing them to reap the benefits of automatic installation of software packages. Doing so will also provide an additional benefit: the ability to dynamically load binary libraries into Tcl.

This technology should also be applicable to Java applets [2], providing additional security for complex applications at little overhead. Rather than authenticate all applets, requiring a relatively expensive check for each small piece of code, our system requires only that a root object be authenticated. Once this is done, the integrity of the objects immediately below the root is ensured because their names are embedded in the authenticated objects. This can transitively be applied to the entire dependency graph, allowing a computer to check most applet code locally without relying on external certificate providers.

7 Conclusions

This paper has presented a Tcl package, `tcdn`, that allows Tcl developers to create distributions of their code that have several advantages over current Tcl distribution methods: freedom from version conflicts, integrity checking for packages, and the ability to dynamically download needed modules from remote sites. It is our hope that this package will enable Tcl-based applications to reach a wider audience by simplifying the installation process as well as the upgrade process. All that is necessary to install an entire application is the content-derived name of its root object and a location from which to get it; from there, everything is handled automatically. If the software is upgraded, the user need only get a new root object from the developer, and the package dependencies are updated automatically.

Because `tcdn` provides integrity checking and the ability to fetch missing packages from remote server sites, we believe it will be essential for developers who wish to make Tcl software available via the Web. By providing both integrity and ease of use, `tcdn` enables even novice users to run complex Tcl applications without the need for complex installations or the fear of trojan horse packages.

Code Availability

Further information about content-derived naming is available on the Web at:
`http://www.csee.umbc.edu/~elm/Projects/CDN/`.
This page contains references to other work on content-derived names as well as the Tcl source code and documentation for `tcdn`.

References

[1] *Secure Hash Standard*, FIPS-180-1, National Institute of Standards and Technologies, U.S. Department of Commerce, April 1995.

[2] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, 1996 (Addison-Wesley).

[3] A. van der Hoek, R. S. Hall, D. Heimbiger, and A. L. Wolf, "Software Release Management," CU-CS-806-96, University of Colorado, August 1996.

[4] J. K. Hollingsworth and E. L. Miller, "Using Content-Derived Names for Configuration Management," 1997 Symposium on Software Reusability (SSR '97), Boston, MA, May 1997.

[5] J. W. Moore, "The Use of Encryption to Ensure the Integrity of Reusable Software Components," International Conference on Software Reuse, Rio de Janeiro, November 1994, pages 118-123.

[6] R. L. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321, Network Working Group, April 1992.

[7] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," Proceedings of the Winter 1996 USENIX Conference (San Diego, CA), January 1996, pages 33-44.

Using Tcl to Rapidly Develop a Scalable Engine for Processing Dynamic Application Logic

Greg Barish
Healtheon Corporation
barish@isi.edu

Abstract

At Healtheon, we used Tcl to rapidly develop a scalable, high performance rule engine for processing dynamic application logic. The nature of our application requirements, plus the challenge of delivering robust software in a timely manner made Tcl an optimal overall choice in our deployment. We were able to improve rule processing performance by careful language construction and support for concurrent execution. We developed a mechanism for implementing data-driven language extensions, called rule concepts, which allowed us to present a customized language for each client, and encouraged rule reusability. Our experience with using Tcl in our application system was also representative of software engineering choices that small companies often make in pursuit of rapidly developing a well-balanced system solution.

1 Introduction

At Healtheon, we used Tcl as the basis for the implementation of a *rule engine*, a mechanism for processing dynamic application logic. Our design, implementation, and deployment were notable because these phases revealed techniques for integrating Tcl in environments which must meet the scalability and performance demands of an on-line service.

In this paper, I initially describe the problem domain and present how Tcl addressed the basic application and system requirements. Next, I present the evolution of our extensions (our rule language). This serves to illustrate techniques for addressing the performance of run-time interpreted languages and suggests a data-driven solution for implementing dynamic extensions. Moreover, the evolutionary description represents a case study for why the adaptability of an extensible third-party interpreter resulted in a more economical solution than having implemented a custom rule processor ourselves. Finally, I discuss various system integration and deployment approaches towards improving the availability and concurrency of a Tcl-based application service, outside of modifying the interpreter itself.

An important sub-theme of this paper is one of software engineering. In particular, I refer to the "rapid rate" of development and deployment of our Tcl-based rule engine as a feature. Since we are a small company, quickly deploying reliable software is a constant demand, and I imply several times why Tcl was uniquely qualified to address not only the basic unit requirements, but was also the best overall solution in terms of system integration and robust operation.

All too often, software are solutions are heralded only for their elegance at handling a single complex problem. Less commonly rewarded are software solutions which successfully meet several software engineering demands, in addition to achieving basic unit functionality. These demands include the ease of integration, method of maintenance, and time and resources required for development. They can be referred to as economical features of software development. A novel aspect of our use of Tcl was how well we found it at addressing these broader issues and thus providing us with the best overall solution.

1.1 The Healtheon Benefit Manager

At Healtheon, one of our key applications is a on-line benefit management system, called Benefit Manager, which employers can use to process employee benefits. Administrators use the system to choose and customize benefit packages and plans for their employees. Subscribers use the application to make and update their benefit choices. These bulk of these selections are usually done once a year, during a series of weeks known as Open Enrollment.

Since Benefit Manager exists as an on-line Internet service, with concurrent sessions potentially numbering in the tens to hundreds of thousands, it must also wrestle with requirements for availability, scalability, high performance, and fault tolerance. These demands are also true of other applications employing the on-line service model, such as America On-Line, Yahoo, and Amazon. In this sense, Benefit Manager is representative of a model of application deployment becoming increasingly common in commercial, transaction-based systems.

1.2 Benefit Rules

One of the key application requirements of the Benefit Manager application was the need to support the encoding and processing of *benefit rules*. Benefit administrators at a given company specify these rules and associate them with benefits, plans, and selections for various types of employees. Rules figure such things as benefit and plan eligibility, plan costs and credits, plan dates of effectivity, validation of selection combinations, and several other such determinations.

Rules are run on behalf of a given subscriber and are frequently a function of some demographic, employment, or existing benefit attribute associated with that subscriber. For example, a certain medical plan geared towards retirees might have an eligibility rule based on the employee age. Another example would be a case where a legacy benefit plan is only available to subscribers who are already enrolled in that same plan for the current benefit period (a rule based on existing selection information).

However, there are also rules which are partially based on the particulars of a specific transaction. For example, an employee may be eligible for medical plan *M* and dental plan *D*, but not eligible for the selection of both of them. Thus, such rules are run based on the session data associated with the transaction.

Processing rules always involves the generation of *rule results*, which vary based on the type of rule. For example, eligibility rules return true/false results, cost rules return dollar amounts, and effective date rules return calendar dates. While most rules typically return a single result, some do return multiple results, and some even return an unbalanced number of results (i.e., one result on success, two results on failure - the second result being the error message).

At Healtheon, rules are expressed in a rule language. They are associated with the appropriate benefit, plan, or other relevant object in our data model. Therefore, a many-to-many relationship between employees and benefit information exists and thus most rules are shared by large groups of employees.

Typically, making benefit selections during Open Enrollment results in between 100 and 150 rules being processed per subscriber. However, although rules are shared, employers will still have a high number of total rules on the system, to provide coverage for all types of employees and support all possible benefit and plan combinations. It is not uncommon for a typical employer of 10,000 employees to have over 500 rules on the system. Rule population varies by company and is more of a function of their existing benefits complexity rather than the company size.

1.2.1 Pre-Computation

One might be tempted to ask why rules could not be pre-computed, to remove the burden of run-time interpretation. Generally, it is not practical to pre-compute rules. For one, rules are often date-sensitive. Consider a plan eligibility rule such as: "*a subscriber is eligible for plan if that subscriber has been in the same plan for at least two years*". If that rule is pre-computed a month before a subscriber actually tries to choose that plan again, it might obviously return different results.

It could be suggested that rules be pre-computed on a daily basis, perhaps during times when the system is relative idle and better able to engage in such CPU-intensive activity. This is still impractical when considering the workflow of a benefits enrollment. Frequently, subscribers will use Open Enrollment to first change demographic data *before* selecting benefits. Thus, the same session is often used for both tasks. Sudden changes in personal information would therefore often cause the pre-computed rules to be invalid.

It is conceivable that a dependency graph could be constructed such that only those rules which need to be re-computed after such scenarios actually are, but leads to drastic increases in solution complexity. It would have required a sophisticated parsing (actually, compilation) technique to be employed whenever rules were updated in the system. Moreover, this effort implies that a solution for detecting indirect requests for data retrieval be implemented. This is analogous to the problems associated with detecting pointer-based references, such as that which is done during data flow analysis for purposes of compilation.

2 Choosing Tcl

We decided to use Tcl as the basis for our rule language for two major reasons: (a) the effectiveness with which it fulfilled our basic application and system requirements and (b) that it was, from a software engineering perspective, the most economical of the solutions available.

During this discussion, it is useful to also relate observations which led up to our decision, since it illustrates why a language like Tcl becomes attractive to a small company which needs to rapidly deliver reliable software.

2.1 Application and System Requirements

As described earlier, the most serious application unit requirements we had on the interpreter was that it needed be easy to extend and would contain support for moderately complex data structures and their associated data manipulation functions. Obviously, Tcl met both of these demands easily.

System-level requirements included those of attaining high performance, scalability, and availability. Although the choice of interpreter and language design did have some impact on these challenges, it later turned out that it was better to deal with these issues at a higher level of software - the rule engine itself. I describe these approaches in later sections of this paper.

Still, an important additional system-level requirement to address at this point had to do with the ease of integration. As a small software company, we were concerned about how development and runtime system would be affected by incorporating Tcl. While using third party software can be attractive for how much it can enhance an existing system, difficulty related to integration often results in subtracting away any profit which was made.

For example, we were immediately concerned about performance issues associated with parsing. We were also questioning how we our use of multithreading might impact Tcl. Other issues included: integration with C++, integration issues with CORBA (our choice for distributed application development), API portability, effect on system security, and the time required for integration.

2.1.1 Code Maturity

Many of our concerns about integration had to do with how comfortable we felt trusting the code. We did not want to build and deploy a system only to find out that we were plagued by problems with a third-party API.

In addition to being one of the most prolific, freely available interpreted languages, Tcl had been around for several years. We knew that it was heavily used and extended by others in the industry. We felt that there was an excellent chance that the code would be stable and efficient.

Even more attractive was the fact that we were not simply integrating with a code library - we had access to the source code itself. Thus, during debugging, we were able to ensure that various bugs were definitely not Tcl-related.

It should be noted the opposite is nearly always true when dealing with third-party code which has been purchased. In those scenarios, developers have to deal with support teams which look at various potential bugs on a case-by-case basis. In terms of efficient software engineering, this is highly undesirable in terms of both time and cost. Even worse, though the developers typically have already paid large amounts of money to use commercial third-party software, they often then have to pay additional fees for support!

This is not to suggest that the universe of industry software engineering problems would be resolved if companies just gave away their code. Tcl is unique in this respect, and the above observation is merely a testament to this specific case.

2.1.2 Simplicity of Extensibility

There was a minimal learning curve associated with figuring out how to extend Tcl: only one API call was necessary. Understanding the related data structures and requirements involved in writing an extension were also very simple. We successfully tested an extension the same day we downloaded the software.

Later, we also found great value in the ability to remove commands from the language (see *Security*, later in this section).

2.1.3 Portability

At the time we needed to choose a rule language, it was unclear whether our system would be running under Sun Solaris or Windows NT. That Tcl had been ported to several platforms made this concern a low-priority issue during consideration and allowed us to continue development on the rule engine in parallel, without delay.

Furthermore, in looking at the code organization, we could see that the operating system dependent code was well distinguished from the generic code, so we felt that any modifications or platform-specific enhancements we might have to make on our own would be an easier task than usual.

For example, our platform team had come up with a lightweight, portable thread library [Kougiouris97] which we thought we might need to incorporate somehow in the interpreter itself. Given the distinction of operating system dependent code, we felt it would be easier than usual to know what parts of Tcl might be affected and where to find the related code.

2.1.4 Security

Initially, it was unclear who would be coding the rules we needed and how it would be done.

Would they be coded from clients through a Web interface? If so, then what kind of impact could forged rules have on our system, in addition to destroying the correctness of the rules themselves? Would clients be composing rules through a graphical editor (which would more or less guarantee correct syntax – at least there would be a finite bound on the range on input we would be receiving) or did we have to worry about the horrors of handcoded rules (which might bring along things like infinite loops)?

Most of these questions would have become serious issues had we not been able to disable parts of the existing API. The fact that we could take out some of the control flow commands, like `for` and `while` - which we didn't need for our rules – allowed us the best of both worlds. Thus, in addition to our own command set, we could still harness Tcl for parsing, variable support, and mathematical functionality, without having to worry about the risk of commands with potentially serious consequences, commands we didn't need.

2.2 Software Economics

It is important to emphasize that we did not view Tcl as necessarily an optimal rule language solution. Certainly, a more attractive scenario would be one in which we built our own custom parser, or even more attractively - our own rule virtual machine. Then, we could pre-compile the rules and remove the parsing element altogether. However, such tasks require significant investments in additional time and development staff. As a small company, we could not afford to make that kind of investment.

Furthermore, there was additional risk involved at investing in an a more optimal solution. More time spent on developing a complex solution would result in less time for system integration, meaning that overall system stability and robustness would be sacrificed. Even more disturbing was the fact that, as mentioned earlier, rule language requirements were volatile. Building extensibility into our own parser or virtual machine would be more complex, require an addition investment in time and resources, and thus increased risk.

What was very obvious to us was that using Tcl gave as a near-optimal language solution at a fraction of the cost. We predicted (and were later proved correct) that building a rule engine based on Tcl would require minimal investment in time and development staff, and would lead to more time to be spent on system integration, thereby increasing our ability to address system robustness. At Healtheon, we felt that even the most sophisticated and cutting edge unit technology looked poor if the entire system it is integrated into does not hold together well. The whole was not simply the sum of its parts.

3 The Rule Engine

Before discussing the rule language further, it is worthwhile to first understand the approach we took towards designing the rule engine, the mechanism which housed the Tcl-based rule interpreter.

The Benefit Manager rule engine provides a simple, lightweight API for processing benefit rules. The engine exists as a C++ class library which Healtheon CORBA-based application servers can link with at compile time. Applications merely make a single call to the `interpret()` function when they want to process a rule. The rule engine eventually calls `Tcl_Eval()` to process the logic and then returns a list of zero or more rule results back to the application.

One of the novel aspects regarding integration of the rule engine with Tcl was how easy it was to gain access to important rule engine data structures during the course of interpretation. It is easiest to understand this profit by way of example. Recall that one of our rule processing requirements was the ability to send back rule results after an interpret. One of the extensions we made for this turned out to be the command `EligibleWhen`, which would return a boolean result of true if the list of arguments passed to it were all non-zero. A sample rule which used this command would be:

```
EligibleWhen [IsMarried];
```

Figures 3a, 3b, and 3c show key parts of the Rule Engine related to this example extension. They reflect three of the important integration stages: (i) runtime initialization of commands (in the Tcl interpreter) based on data structures automatically generated from our data model, (ii) when the call was made to `Tcl_Eval()`, in the course of rule processing, and (iii) the implementation of the extension itself - the code which actually performs the logic and stores the rule results.

```
TCM TclCmdMap[] =
{
   ..
   {"EligibleWhen",
    RuleLanguage_EligibleWhen}
   ..
}

int
RuleMotor::initialize()
{
   ...
   ...
   for (int i=0; TclCmdMap[i].fn!=NULL; i++) {
       Tcl_CreateCommand(
           m_interp,
           TclCmdMap[i].name,
           TclCmdMap[i].fn,
           (ClientData)this,
           (Tcl_CmdDeleteProc*)NULL);
    }
   ...
   ...
 }
```

Figure 3a: Initialization

In particular, Figure 3a shows the simple loop used for creating commands upon Rule Engine initialization. The important part of this phase is to note that a subset of the commands (those tied to the data model) were automatically generated by the data model DDL itself, which allowed us the flexibility to change our model without having to, say, rewrite lexer rules every time. This process is described further later in this paper, in section 5.1.

```
RuleResultList*
RuleMotor::interpret(
   char* a_rule)
{
   m_ruleResults->clear();
   ...
   ...
   int code = Tcl_Eval(m_interp, a_rule);
   ...
   ... During eval, results will accumulate
   ...
   return m_ruleResults;
}
```

Figure 3b: Access

Figure 3b shows the integration point with Tcl during rule interpretation. Rule Engine worker threads, called Rule Motors (described later, in section 6.2), make the actual call to `Tcl_Eval()` themselves and keep private data structures for rule result aggregation.

```
int
RuleLanguage_Tcl_EligibleWhen(
   Tcl_Callback* a_ptr,
   int argc,
   char** argv)
{
   RuleMotor *theMotor = (RuleMotor*)a_ptr;
                   ..
                   ..
      (analyze arguments, determine T or F)
                   ..
                   ..
   if (noneAreFalse) {
      theMotor->appendBooleanResult(TRUE);
   }
   else {
      theMotor->appendBooleanResult(FALSE);
   }
}
```

Figure 3c: Processing

Finally, Figure 3c shows how the example command acquires the pointer to the worker thread which is running the rule (the particular Rule Motor) , performs its necessary logic, and then associates the result of the rule with the motor responsible for rule invocation.

Notice that a pointer to a rule engine data structure is specified when making the `Tcl_CreateCommand()` call in part (i). During runtime, (ii) is invoked to perform the rule processing. Then, during interpretation, the data structure from (i) can be accessed by casting the callback pointer upon entry to extension, as shown in (iii). Upon return in (ii), the modified data structure can be analyzed. This was a major asset at runtime, since it prevented us from having to marshal data between steps (ii) and (iii).

There are several other important issues related to improving performance, availability, and scalability of the rule engine in terms of deployment. However, before discussing these issues, it is first useful to understand the nature of the Tcl extensions we made, in other words, the rule language.

4 Rule Language Design

It took us two iterations to arrive at an effective rule language. Both are worth describing because they illustrate how Tcl became so valuable to us while our application system matured. The first attempt saw us make only a few extensions to Tcl, enough so that we could retrieve information from our database and process rule results. The resulting language was sufficient but problematic.

The second attempt was far more effective because, by then, we had a much clearer picture of the rule requirements. It gave us time to look at how clients wanted to encode rules and adapt the language accordingly. Again, it should be noted that had we not used Tcl, the prospect of iterating the rule language as we did would have been far more risky and less likely to succeed.

4.1 Take One: The Minimalistic Approach

The initial version of the Healtheon rule language included less than 10 commands, each of which was essentially a generic mechanism for addressing the language requirements.

For example, we had one command (`DbGet`) for retrieving subscriber information from the database. This extension took a table and attribute name as arguments, issued a dynamic SQL query to our database, and would return the associated values for that subscriber.

We found numerous problems with this approach:

- **data model exposure**: In order to specify the arguments of a `DbGet`, the user was obviously required to understand our data model. Exposing a data model is generally not good practice - the system should present a consistent interface to all users for the long term. Exposing the low level details of the model hampers the ability to change the model.
- **command usage ambiguity**: If the table name could be any one of our tables, how could we enforce that the author specified the correct number and type of table keys in order to resolve a unique row? Again, this would clumsily expose the underlying data model. Even worse was that it made the `DbGet` command more ambiguous - it was unclear how many arguments it needed for a given call.
- **greater parsing demands**: We had to assume that clients might enter table names and attributes in

varying styles - wrong case, slight misspelling, etc - and we needed to determine how to handle such input. Furthermore, there was simply more to parse: having a generic data retrieval command such as `DbGet` implies that additional parsing will need to be done to figure out exactly what to get.

- **manual data typing**: Tcl is typeless, so after getting the data back, the value essentially became a string. Any other function which took the result of a `DbGet` as an argument would have no idea what kind of actual type it was dealing with, unless it was overtly specified by the author.

  For example, consider the problems with comparing two dates - the subscriber's hire date at the company and his birth date. Although we would want to say:

```
if [Compare LessThan
      [DbGet Subscriber BirthDate]
      [DbGet Subscriber HireDate] ]
then
    ...
endif
```

  this would be problematic unless the type information was included:

```
if [Compare date LessThan
      [DbGet Subscriber BirthDate]
      [DbGet Subscriber HireDate] ]
then
    ...
endif
```

These issues are representative of those associated with implementing the other generic extensions we did in our initial language definition. We found that they placed a heavy burden on the rule author, they are far more prone to error, they incurred greater parsing demands, and they were overly verbose. Perhaps the greatest crime was that rule author, the most valued client of all, was presented with an ambiguous, cumbersome interface.

Despite all of these problems, there was one important benefit to our initial version of the language: the speed at which we could implement a functioning rule processor. The rule engine became one of the first modules of the alpha portion of the application software to actually work. This milestone allowed the engineering team to focus more on system integration issues and re-assign developers to other projects which were not yet completed. Additionally, it allowed us to begin coding rules for some of our initial clients.

4.2 Take Two: The Final Cut

With proof of concept under our belt, the second version of Benefit Manager saw us revisit the design of the rule language. By this point in our development, we were now made clearly aware of the problems associated with implementing generic commands. For the second release, we wanted less verbose rules, less parsing, decreased burden on the rule author to understand various uses of the same command, and a syntax which generally left fewer opportunities for user error.

4.2.1 Improving Data Access

Early in the process, we attained a major milestone by developing a new paradigm for data retrieval. Our approach was to create a Tcl extension for every possible attribute of every table in our retrieval domain. The name of these commands was the concatenation of the table and attribute for that item.

Implementing each potential retrieval as an extension also gave us the opportunity to hide the data typing associated with that item. For every database command, our rule language supported two styles of execution. In the first, no arguments were specified and the string (typeless) value of the attribute was resolved. The second form required comparison arguments and returned either true or false, depending on the result of the comparison. It was this second form of data access and comparison which gave us the opportunity to implement *automatic datatyping*.

Figure 4a shows the how a rule based on subscriber salary would be authored in the first and second versions of the rule language. Obviously, the second version is more compact, less cluttered, incurs less runtime parsing, and makes the data typing automatic and transparent.

Version 1:

```
if [Compare double >
      [DbGet Employee Salary] 50000 ]
then
    ...
```

Version 2:

```
if [EmpSalary > 50000] then ..
```

Figure 4a: Improving Data Access

4.2.2 Improving Rule Result Reporting

Another major language improvement was to make rule result reporting easier. As was the case with data access, the first version of the rule language placed the burden of declaring the rule result type in the hands of the author. Even more concerning was that authors might forget to return a result (leaving the application confused about the status of rule satisfaction) and the high potential that certain control flows might not lead to any results being reported.

To address these issues, we devised specific styles of commands which would encourage full reporting. The heart of the problem was the reliance on `IF-THEN` logic. Such logic was characteristic of most rules and we attempted to implicitly capture that logic in our new commands, improving code compactness and removing some of the potential for coding errors.

Figure 4b shows how an benefit eligibility rule appears in both versions of the language. Obviously, the first version contains more code and places a higher degree of responsibility on the author in terms of control flow checking. The second version makes the `IF-THEN` logic implicit in the `EligibleWhen` command. This command simply takes a space delimited list of values and returns false (i.e., "not eligible") if any of these values are zero.

Version 1:

```
if [Compare double >
      [DbGet Employee Salary] 50000]
then
   ReturnRuleResult boolean true;
else
   ReturnRuleResult boolean false;
```

Version 2:

```
EligibleWhen [EmpSalary > 50000];
```

Figure 4b: Improving Rule Results

There are some other worthwhile observations to make about the effect of the language metamorphosis. These included the number of commands in the language: it did require more work on the author's part to know what the categories of commands were. However, we felt that this was not unusual for a scripting language. Also, many of the command names were predictable.

Another related, but subtler, aspect was the level of redundancy in the new language. Recall that we had to process several types of rules: eligibility, costs, credits, dates of effectivity, etc. Now, while there were close to 10 types of rules to deal with, we were always returning a single result, the type of which was either boolean, double, or date. This implies that we really only needed three distinct commands for returning rule results.

While critics might lobby for one mechanism for returning a type of double, there was an obvious subtle benefit to having distinct commands on a rule type basis: increased language transparency. For example, we were freer to update the semantics of eligibility instead of forever remaining tied to the notion that it only represented either True or False.

By making these changes to the rule language, we improved the performance of rule interpretation, decreased the potential for error, and hid the ugliness of data typing. Most importantly, from the authoring point of view, the rules were much easier to read and understand. This, in turn, made them easier to author.

4.2.3 Rule Concepts: Dynamic Extensions

It is often desirable to customize a language to best meet the specific needs of a client. Each company has its own way of doing business, its own *business logic concepts*, which frequently play a role in rule processing. Our goal was to do what was possible to support the expression of rules in these simple, familiar terms, to improve the usability of the rule language, as well as to promote rule compactness.

Our solution was to support the declaration of *rule concepts*, company-specific extensions to the rule language. In this sense, the rule language was thus a union between the base language and the concepts for a given company. In practice, rule concepts were simply macros to facilitate the simple expression of a awkward or complex computation.

Suppose a company is based in California and frequently uses the state tax rate in their benefit plan cost rules. They might want to express a cost rule as something on the order of: "*the cost of this plan is $200 plus 1% of the employee's salary multiplied by the state tax rate*". We might want to capture "tax rate" as a concept. To do this, they would use our administration interface to name a new language command called `TaxRate` and define its meaning. Optionally, if the phrase "*one percent of <some number> times the tax rate*" was a frequently used computation, the company could even define that as a concept (i.e., `OnePercentAndTaxOf`),

The benefits of rule concepts are three-fold: they provide a key level of functional abstraction in the language, they increase the re-usability of rule code in the system, and they also allow us to personalize Tcl to best address the specific business language of our clients.

5 Language Integration

The new version of our rule language was also notable in terms of the issues it raised related to system integration. Some of these had to do with the management of having over 150 Tcl extensions, specifically in terms of proper maintenance and namespace clashing. Other issues involved the relationship of the language to the application data model and how useful application-level data objects might be represented .

5.1 Automating Generation of Tcl Extensions

With over 100 commands for data access alone, there was concern about the ability to manage the development of these extensions without assigning more programming staff to the task. Since the nature of all of these extensions were the same (retrieve data, optionally provide comparison logic), we decided to spend time developing a mechanism for automatically generating the code for these extensions. Our process for this is shown in Figure 5a. The typical cycle of development was to first update the data model as required, export the associated data definition language (DDL) to a file, use the DDL to the automatically generate source code for the Tcl extensions, and finally rebuild the extensions library. This level of automation allowed us to efficiently adapt the data model as required. In fact, DDL revisions became a far more problematic issue for the applications themselves (which often referred to attributes and tables literally) than it was for the rule engine.

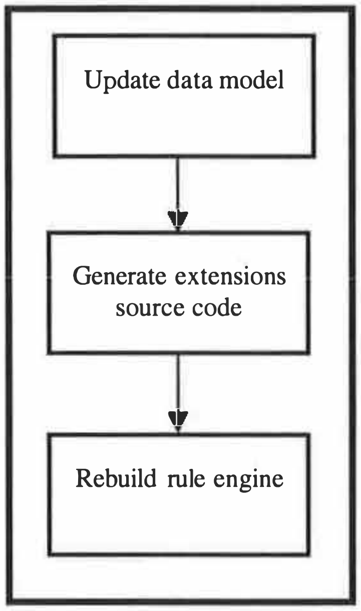


Figure 5a: Code Generation Methodology

5.2 Naming Issues

One additional issue we encountered was that of the command namespace management. For example, all of the tables which could be data access candidates in rules contained attributes which were named the same, but included in different tables for various reasons (they were foreign keys or just deliberate duplicates of information for purposes of security via data partitioning). What this meant, for example, was that we could have a field in the demographic table called "`Birth_Date`" and a field in the employment information table called "`Birth_Date`". We needed to mechanism to establish closure [Neuman89], to ensure name uniqueness.

We took the common route of prepending a unique (table-based) prefix to each command. Thus, a rule for comparing birth dates would appear as:

```
if [EmpAge == [BenAge]]
then
    ...
    ...
```

This would compare the age of the subscriber as listed in the employment (`Emp`) information with the age of the subscriber in the benefits (`Ben`) table.

There is additional naming complexity when considering how to deal with implementing rule concepts. For example, if two companies want to create a concept called `ComputeTax`, how should this be resolved? Should the company who first requested this concept be awarded the right to name it? More alarming was the privacy aspect of concepts: since the same rule interpreter was used by all parties, did this mean that a concept designed by one company would be accessible by all other companies? Namespace collisions are common problems when using Tcl, mostly due to the lack of scoping in the language [Libes95].

Our solution to this problem was to implement *run-time dynamic scoping*. The basic idea was that the Tcl interpreter would consist of the normal extensions of the rule language, plus the union of all concepts (company-based extensions). This meant that, even though two different companies declared their own version of a concept like `ComputeTax`, there was only one extension made. At runtime, a hash table of concepts was maintained in memory. Thus, for `ComputeTax`, the hash bucket associated with that entry would contain a linked list of two elements, representing both versions of the concept.

The methodology for concept resolution - or closure - was simple: since rules are always run on behalf of a subscriber, simply determine which company providing benefits for that subscriber and use that company identifier to choose which concept to execute.

5.3 Data Model Aliasing

As described in 5.1, we generated our extensions based on the data model itself. However, we remained concerned about the potential for legacy rules - ones which were no longer valid after a data model update. For example, if a client had written a rule like that in Figure 4b (version 2) and we suddenly decided to rename that attribute to "`AnnualSalary`", we would indirectly invalidate an existing rule. The lack of a mediator between the rule language and the data model remained problematic.

To address this issue, we devised a map-based mechanism for ensuring data model transparency. The methodology consisted of a scheme in which multiple names, known as *aliases*, could refer to the same attribute. Thus, if an attribute name was changed, we could simply amend the map to include support for the old name by providing an alias, without modifying the actual company rules themselves. Multiple Tcl extensions would therefore be created for the same attribute, but only one extension would actually be implemented. The rest were simply pointers to this code. The notion of aliases is not new, it is simply a variation on an implementation of symbolic links [Lampson85].

Obviously, while aliases succeeded in resolving the issue of attribute name changing, they did not help out when an attribute was removed from the data model. This is a more complex problem and one where a resolution may not be possible.

6 System Integration and Deployment

The underlying platform software at Healtheon is a scalable distributed object system. Among other features, it contains support for high availability, concurrency, fault tolerance, security, and naming. In this part of the paper, I discuss techniques for the integration and deployment of the rule engine such that these system features could be exploited in an attempt to improve scalability.

In particular, I show describe our experiences with improving rule engine availability and performance. While the former was not successful, the latter was very successful, and led to increased scalability of the resulting software upon deployment. While these experiences only marginally related to Tcl itself, they represent ways in which Tcl can be integrated into systems to both address general reliability as well as to counter performance problems associated with run-time interpretation.

6.1 Improving Rule Engine Availability

As described earlier, building an application in our system which used the rule engine requires linking with the associated library.

Initially, we had wanted to improve the availability of the rule interpreter by wrapping the rule engine in a CORBA shell and allowing an object request broker (ORB) or separate load balancing tool to launch the number of rule engine instances necessary to deal with given client demands. This would have prevented a misbehaving, frequently crashing application from also destroying access to the rule engine for other servers.

While this would have improved availability and fault tolerance, it would have also severely hampered performance. Recall that processing a set of selections for a given subscriber results in over 100 rules being processed. If the rule engine existed as a distinct server, this would lead to an unacceptable number of network calls.

An alternative, slightly more optimized approach would have been to simply combine a CORBA module for the rule engine with the other modules for the Benefit Manager application in the same server. When multiple modules exist in the same server, a form of optimized, lightweight RPC [Bershad90] is often used. This results is what is essentially a local function call.

While this reduces the data copying and network overhead, it does not address the needless marshalling of data between module boundaries. Since the workflow of rule processing is such that one segment of C++ code (the application) is calling another (the rule engine), the arguments passed to the rule engine must thus be marshaled into CORBA types, even though these arguments are never even transmitted with a networking protocol.

In the final analysis, we stuck with our original idea of using a C++ class library as the basis for the rule engine, as the sacrifices to be made for improved availability were not worth the price is decreased performance.

6.2 Concurrent Rule Execution

On a more positive note, we did have far more success with improving performance by increasing rule processing concurrency. Specifically, we made improvements to the rule engine itself, in terms of thread safety, without needing to investigate a Tcl-based solution for multithreading.

To achieve acceptable concurrent processing, we designed the rule engine to support pools of *rule motors* - each essentially a wrapper around a rule interpreter data structure - several of which would be available when rules needed to be processed. Thus, each motor supported its own `Tcl_Interp` data structure, as depicted in Figure 6a.

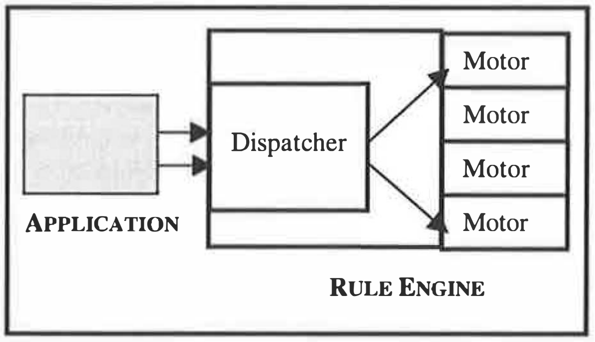


*Figure 6a: Concurrent Rule Processing Via Rule Dispatcher Each Rule Motor contains a Tcl_Interp**

When handling requests for rule interpretation, the rule engine motor dispatcher would grab the process mutex, search for an available motor, hand off the request to that motor, and mark it as taken. In the event that all motors were occupied, the rule engine would simply wait for a broadcast signal (sent when a motor was done with processing a request) and then choose that motor as the destination for the incoming request.

It should be noted that there was no need to support multithreading for reasons of correctness in processing, since the internal Tcl data structures which cannot support concurrency (the global variables) are only related to those Tcl commands for file and process management, commands which we did not support with our rule language.

However, multithreading was important for purposes of supporting high concurrency and thus improved system throughput. Simply using a thread-safe version of Tcl would not have been an optimal solution, since what was really needed was for the entire engine to support concurrent access. Therefore, maintaining pools of interpreters was a more reasonable solution to the problem.

7 Discussion

Employing Tcl as the foundation for our dynamic logic engine ultimately proved to be successful. We were able to rapidly construct a powerful rule language, through a combination of our own extensions as well as leveraging existing Tcl functionality. The ease at adding and removing language commands allowed us to write code generators and easily adapt to rapidly changing design requirements with minimal integration.

The rate at which we were able to develop the rule engine allowed us to reallocate our resources and spend more time improving system performance and overall integration. If we had simply devoted several months to the development of our own parser or pre-compiler, we would have not been able to adapt to the moving target of volatile design requirements, all too common at a small company.

There were some disadvantages to using Tcl, but most of them were expected. We ran into some performance problems, some due to our own parsing and some related to the Tcl parser. We almost certainly could have achieved better performance through a more lightweight, application specific parser or even rule virtual machine which would process pre-compiled rules. However, these would have been impractical solutions and would have forced us to sacrifice crucial aspects of overall system integration.

At a larger company, development teams have more time to approach milestones, and can afford to spend long periods of time in the design and prototyping phase. Also, larger companies tend to deliver a tool they think addresses a market demand and then tackle customer requirements in future releases. In short, they can easily push products into the market channel.

In contrast, a small company cannot afford to simply throw software out onto the market. It needs to be highly sensitive to the requirements of its core customers, as much of a moving target as that can be, while still delivering a well-balanced product: it thus becomes essential to deploy something quickly, but which still ensures correctness and robustness, despite changing requirements. Using Tcl allowed us to meet the necessary requirements, quickly deploy advanced functionality, and to spend more time improving system integration.

8 Acknowledgements

I wish to thank the following people at Healtheon for their continued input and support related to the care and feeding of the rule engine: Mohammad Alagebandan, Giamma Clerici, Kittu Kolluri, Shankar Srinivasan, and Theron Tock.

9 References

[Lampson85]	Lampson, B., "Designing a global name service". In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing*, August 1985.
[Libes95]	Libes, D., "Managing Tcl's Namespaces Collaboratively", *Proceedings of the Fifth Annual Tcl/Tk Workshop '97*, Boston, MA, July 14-7, 1997.
[Neuman89]	Neuman, C., "The Need for Closure in Large Distributed Systems". *Operating Systems Review*, 23(4):28--30, October 1989.
[Kougiouris97]	Kougiouris, P., Framba, M., "A Portable Multithreading Framework". *C/C++ User's Journal.* August 1997.
[Ousterhout94]	Ousterhout, J., *The Tcl/Tk Toolkit*, Addison-Wesley, 1994.
[Bershad90]	Bershad, B.N., Anderson, T.E.., Lazowska, E.D., Levy, H.M., "Lightweight Remote Procedure Call". *ACM Transactions on Computer Systems*, 8(1):37--55, February 1990.

Using Tcl to Script CORBA Interactions in a Distributed System

Michael L. Miller

[a]*Advanced Micro Devices, MS 608, 5204 E. Ben White Blvd., Austin, TX 78741*

Srikumar Kareti

[b]*Honeywell Technology Center, 3660 Technology Dr., MN 65-2600, Minneapolis, MN 55429*

ABSTRACT

In this paper we present the extensive use of a scripting language (Tcl) to run human readable/editable scripts in a CORBA distributed batch environment. The developed system is the "Advanced Process Control Framework" as outlined in the "APC Framework Initiative" which is a research and development project undertaken by AMD and Honeywell under the support of the U.S. Department of Commerce, National Institute of Standards and Technology. The APC Framework System has been deployed in AMD's fab 25 and is fully functional. The paper discusses the issues involved in developing scripting mechanism which is capable of both interacting with other CORBA components and handling various data structures which are otherwise not addressable by the underlying scripting language. The flexibility and extendibility of the Tcl scripting language makes it easy to extend the core language. The paper also establishes the need for thread-safeness of Tcl. Examples of various data manipulation operations, calls to different CORBA components, and calls that help in synchronization are discussed.

1. BACKGROUND

In order to fully describe how Tcl is being used in our CORBA environment, it is first necessary to give some background into the project and the distributed system that we are developing. Starting in 1994, AMD identified the need for an extension to our current Manufacturing Execution System (MES) that would support deployment of Advanced Process Control applications quickly and easily into our semiconductor manufacturing facilities (fabs). AMD internally developed a functional specification for such a system, which was completed in mid-1995. The National Institute for Standards and Technology (NIST), a section of the Department of Commerce, announced shortly thereafter an Advanced Technology Program (ATP) competition. Under this ATP, NIST would cost-share up to 49% of a research and development project that would further development in the area of MES systems and integration. The NIST program provides multi-year cost-share funding to industry-led joint ventures to pursue research and development (R&D) projects with high-payoff potential for the nation. Its goal is to accelerate technologies that are unlikely to be developed in time to compete in rapidly changing world markets without such a partnership between industry and the Federal government.

AMD, in partnership with Honeywell, a leading control systems supplier, and SEMATECH, the consortium of US semiconductor manufacturers, proposed the APC Framework Initiative (APCFI).

1.1 APCFI Project

The goals of the Advanced Process Control Framework Initiative project (APCFI) were outlined in the program proposal presented to NIST (National Institute for Standards and Technology) in October, 1995. These goals were to: enable effective integration of "Advanced Process Control" applications into a semiconductor fab to improve manufacturing capital productivity, product consistency, and product yields; establish integration technology for multi-supplier

Further author information -
[a] Email: michael.miller@amd.com; Telephone: 512-602-3959; Fax: 512-602-5299
[b] Email: skareti@htc.honeywell.com; Telephone: 612-951-7302; Fax: 612-951-7438

"Plug-and-play" APC applications; and to demonstrate commercial viability of the APC Framework and its components. To sum up, the main goal of the APCFI projects was to develop a system that would significantly reduce the time, cost, and integration efforts needed to deploy APC solutions.

The scope of the APCFI projects includes support for Feedforward and Feedback Run-to-Run control and Fault Detection applications spanning multiple processes and fab tools and utilizing 3rd-party control software, such as Modelware®, Matlab®, Matlab Toolkits, Mathematica®, and LabView®.

In order to validate the design and implementation of the APC Framework, a number of control projects were selected for early deployment into one of AMD's semiconductor fabs using initial versions of the APC Framework.

1.2 CORBA

CORBA (Common Object Request Broker Architecture) is a specification of an "architecture for an open software bus on which object components written by different vendors can interoperate across networks and operating systems" (Orfali et. al., 1996). It is used by the APC Framework to allow the distributed components of the framework to communicate. In specific we used Orbix, IONA's implementation of CORBA to develop APC.

1.3 Overview of the APC Framework

The APC Framework has been designed to work along with a fab's MES (Manufacturing Execution System – in AMD's case, this it WorkStream by Concilium) and CEIs (Configurable Equipment Interface) to provide APC functionality. It is composed of not one large program, but a number of smaller, specialized pieces that work together. The "interchangeable parts" of the APC Framework are called components. These components are analogous to stereo components, where each component is

1) An independently running entity
2) Provides a subset of the overall APC Framework functionality
3) May be provided by a different vendor

The APC Framework standard describes the functionality, interface, and behavior of each component. The central component at run-time is the Plan Execution Manager, which utilizes Tcl and is described in the next section.

2. THE PLAN EXECUTION MANAGER

To support the goals of the APCFI project, it was necessary to develop a system that would be flexible enough to support just about any supervisory-level (e.g. Run-to-Run) control application. To this end,

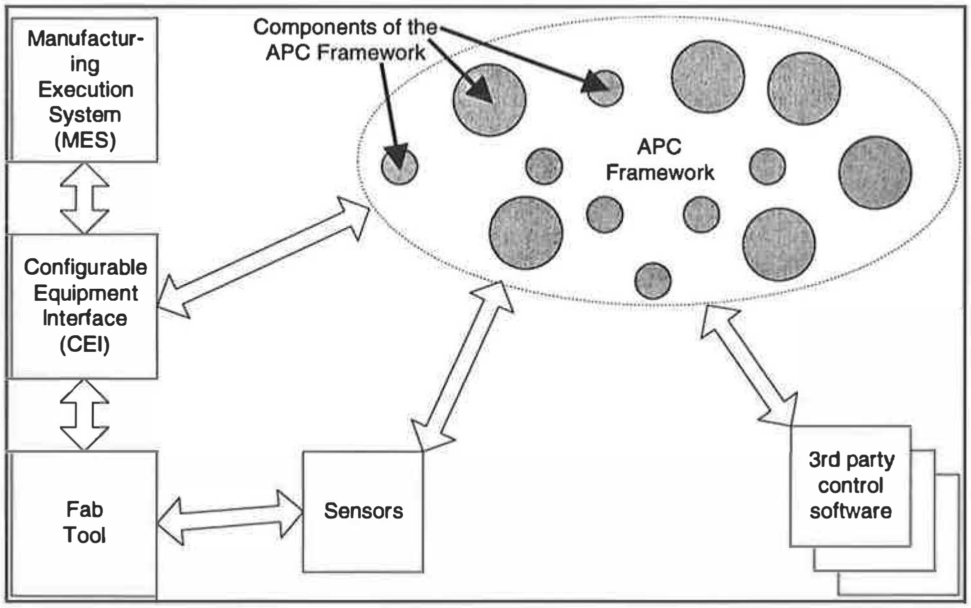

Manufacturing Execution System (MES)
Components of the APC Framework
APC Framework
Configurable Equipment Interface (CEI)
Fab Tool
Sensors
3rd party control software

Figure 1: The relationship of the APC Framework to other software systems.

rather than trying to pre-define a generic sequence of system interactions that would be used at run-time, the project chose to use a scripting approach to allow the maximum flexibility in how the system was used.

The Plan Execution Manager (PEM or PE) component is the "choreographer" of the APC Framework. It is in charge of doing "APC" at runtime for a particular process or metrology tool. To do this, it has the ability to access all of the capabilities of each of the other components in the APC Framework. It executes, or interprets, APC Plans (a collection of Tcl scripts), which specify the actions to be taken before and/or after a lot is processed on a fab tool. The Tcl scripts define not only what the APC Framework does, but in what order they are carried out.

2.1 Why Tcl?

When the project needed to define the scripting language that would be used to drive the system activities at run-time, a number of possibilities were evaluated. Tcl, Perl, and Python were examined, as well as developing a custom language. The latter was quickly dropped because of the rich set of well tested scripting languages readily available and the fact that project resources could be better spent developing other functionality in the system.

Some of the comparison criterion used were: ease-of-use, ability to embed in a C++ application, extensibility, support for CORBA, and support for C structures. The first criterion, ease-of-use, was perhaps the most important. Since the script writers would be programming novices, ease-of-use and a shallow learning curve were critical to the project's success among the users. As the scripting language was to be embedded into a CORBA C++ server, the chosen scripting language had to be able to be embedded into a C++ application. Extensibility was an important consideration, since the chosen language was unlikely to have all of the functionality that would be required. Finally, since the scripts would need to access CORBA functionality and manipulate data in C-like structures, the ability of the language to accommodate these was important.

Python, while it had many features that lent itself to use in an object-oriented environment such as was being defined for the APC Framework, also carried some drawbacks. Foremost amongst these drawbacks was the fact that the scripting language itself is very object-oriented. The users that would become the main users of this scripting language were not object-savvy, and so Python represented a language that would require much more up-front learning on the part of the users before it could be used effectively. The basic Python language, however, did have some support for more sophisticated C & C++ data structures, and was well-suited for use as an embedded language in a C++ application. Python also seemed extensible, although some of the overhead of reference counting, etc., would add some additional work to any extensions being built. Finally, Python did not have any support for CORBA communications in its core, but there was some work ongoing in this area by others.

Perl also had good facilities to support custom user extensions to the language. However, it suffered as well in ease-of-use – the reviewers felt that Perl was syntactically more complex than some other languages, like Tcl, and hence would be more difficult to learn by the script writers. Perl also lacks key facilities which make its use in an embedded application much more complex than either Tcl or Python. Finally, even though Perl seemed to have sufficient capabilities to support some of the more complex data structures that the APC Framework would use, it lacked any CORBA capabilities.

Tcl had an advantage over both Perl and Python in its simpler syntax and hence easier learning curve for new users. Even better was its support for use as an embedded interpreter in a C/C++ application. Since it was developed from the start for use as an embedded language, its facilities were better than both Perl and Python. While there was not any support for CORBA or data structures per-se in the Tcl core language, there were already-developed extensions that would provide this functionality. Even though the project made the decision not to use these extensions in favor of creating our own, the fact that some form of this capability existed already made it easier for the project to roll our own. Overall, Tcl's ease-of-use and embeddability made it the best choice for the APC Framework.

2.2 Use of Tcl in the Plan Execution Manager Component

When the fab process or metrology machine informs the PEM that a specific lot has been brought in for processing, the PEM pulls up a Plan Executor (PE) object to execute the "APC Plan" for that run. The Plan Executor runs various scripts designed by the "Process Engineer" (a.k.a. script writer) and feeds back correction information to the machine to help maintain consistent performance of the machine. The process engineer is typically a chemical engineer and hence it greatly helps to have the scripting language to be readable and English like. There are three types of Scripts: Main Scripts, Sub Scripts, and Event Scripts. These scripts are used in an APC application to define the sequence of actions that the APC Framework performs. The scripts are bundled together in an APC Plan: one (and only one) Main Script, zero or more Sub Scripts, and zero or more Event Scripts. The Main Script is used like the `main` function in C – it is the first script run by the system when it executes a Plan. The subscripts are used to define procedures that the main or event scripts may use. The event scripts are executed in response to certain events should they happen in the system.

When the PE is called to execute a Plan, it begins by creating a Tcl interpreter for the main script in a new process thread. This interpreter is initialized and the APC extension loaded. Next, the PE defines all of the procedures by evaluating all of the subscript files (using `Tcl_EvalFile`), one at a time. These subscripts contain routines common to all the scripts. Finally, the PE executes the main script, one line/command at a time. This is done so that the PE can respond to other requests that may interrupt the execution of the main script between execution of each Tcl command. We allow the user to be able to write the command in more than one line as long as he maintains "Tcl-like" syntax.

While the main script is executing, the PE may receive notification of certain events happening in the system. If there is an event script defined for that event, the PE will execute it in a manner similar to the execution of main scripts. Each of the main script and the Event scripts has an interpreter of their own. Each Interpreter runs in its own process thread and can communicate via shared data and mutex-like locks. This clearly marks the need for a thread safe scripting tool. In the previous versions of Tcl, we were forced to use simple mutex locks around the Tcl library, rather than spending the time modifying the Tcl core to be thread-safe. The latest version of Tcl (8.1) promises to be thread safe, which will make the use of multiple interpreters in separate process threads much easier to use. This is a performance gain for the APCFI system because the locks we used to make Tcl thread safe were very coarse grain. It is not very uncommon to have about five plans running at a time, each in parallel and each of the PEs having a main script and multiple event scripts all running in parallel. The need for thread safeness was high enough to consider porting ptTcl from unix to NT, but the eminent release of Tcl 8.1 made this unnecessary.

When the main script completes, all of the Tcl interpreters are deleted.

3. TCL EXTENTIONS FOR APC

Even though Tcl was chosen to be the scripting language used in the PEM component, in its basic form it did not have all of the functionality needed by this project. Among the added functionality was: Tcl scripts needed to be able to be run in parallel (1 main, multiple sub and event scripts); they needed to communicate data and synchronization information with each other; these scripts needed to create, understand and interpret complex data structures; the scripts needed to communicate to the rest of the world via CORBA; and, finally, since some setup tasks (CORBA calls) might take a considerable amount of time, there was a need to include the capability to run such tasks in the "background" (another Tcl interpreter run in another process thread) and let the calling script continue until it was in need of the data from the background task. Existing Tcl extensions, along with the possibility of building our own, were evaluated. The extensions/modifications to Tcl that supported all of this functionality is discussed below.

3.1 Complications using Tcl

In its core form, Tcl utilizes all data in the form of strings. This makes life simple for the scriptwriter, but causes complications when trying to interoperate in a distributed CORBA environment that uses more complex data structures. Extensions do exist for creating/handling other data structures, but they lacked the ability to handle the CORBA data types that we required. Also, these extensions were built to allow the script writer to construct new data structures "on the fly". While this is desirable in the general sense, the APC Framework utilizes a fixed set of data structures, so this added flexibility is not needed and in fact adds to the learning curve for the script writer. It is for these reasons that the decision was made to write a fixed set of new commands from scratch, which had the added advantage of being able to incorporate features that would allow the data structures to be shared between scripts running in separate interpreters. Our extensions are based on using the `Tcl_SetAssocData` and `Tcl_GetAssocData` calls to store and retrieve the data structures when needed.

In addition, the Tcl core does not have the ability to perform CORBA invocations. We were aware of initial developments extending Tcl to allow CORBA calls, but these packages were either not on the needed version of Tcl or on the necessary platform. Also, as was the case with the data structure extensions, the CORBA extensions provided general facilities for constructing and making a CORBA call to a server.

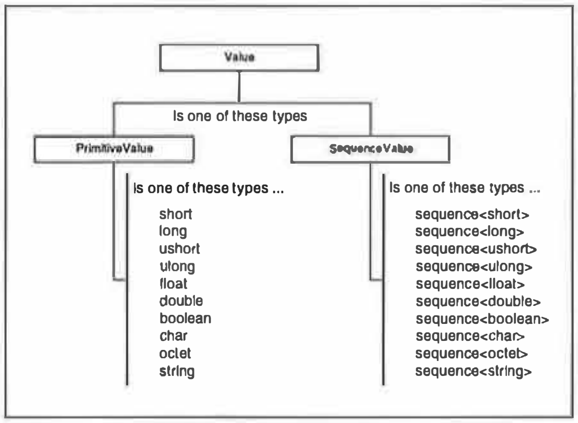
While this provides more capability to the script writer, it carries with it a high price in terms of script complexity. Again, the set of CORBA methods that the script would need to access would be finite and fixed, so this type of general CORBA capabilities was not needed. Finally, we wanted to have a higher level of abstraction where we would make multiple CORBA calls in the same Tcl command rather than have one call to each CORBA invocation. By writing our own CORBA extension specifically for this project, we had that flexibility.

Finally, Tcl has no simple mechanism to support communication between Tcl interpreters running in different threads. In fact, until recently the Tcl core itself was not thread-safe. ptTcl, a multi-threaded version of Tcl, was available for Sun Solaris. It provided not only the ability to launch multiple interpreters in separate threads, but to also communicate with those separate interpreters, However, we did not try to port it to NT due to lack of time and resources. Instead, we built into our extensions the ability to use mutex locks between interpreters and to copy data into and out of a common memory space.

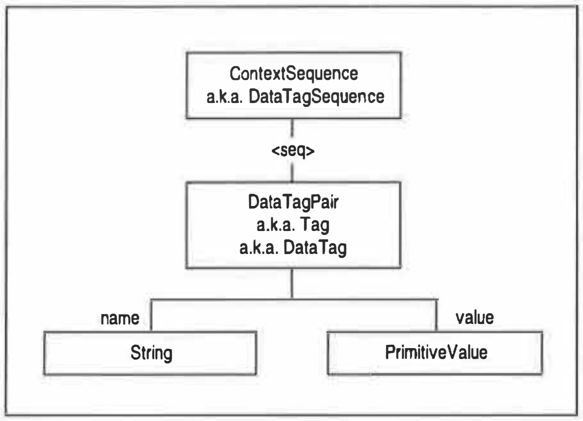
3.2 Data object-related commands

The first additions to the Tcl language made by this project were commands that give the script writer the ability to create, manipulate, and delete all of the different data structures/objects that the APC Framework uses. In general, there are two types of data objects: pure structures and sequences of structures. One new command was defined for each data object type. This command uses the first argument as a switch to define what to do with that object: in general, to create it, get its contents, set its contents, and delete it. A particular instance of a data object is referenced by a unique name – like a variable name. This name, or key, is passed as an argument to each of the new commands.

Figure 2 shows an example of some of the data objects used by the APC Framework – the Value, DataTagPair, and DataTagSequence structures. Tables



2a



2b

Figure 2: Value (a), DataTagPair (b) and DataTagSequence (b) data structures

Command	Arguments	Returns	Comments
`DTPair`			
`create`	`Name Type value DTPkey`		Create DataTagPair
`get`	`DTPkey`	`Name [Prim./ Seq.] Type value`	Return the contents of DataTagPair
`getvaluekey`	`DTPkey Vkey`		Put the contents of the DataTagPairs's PrimitiveValue in a new Value called Vkey
`set`	`Name Type value DTPkey`		Set contents of existing DataTagPair
`delete`	`DTPkey`		Delete the DataTagPair

Table 1: The DTPair Command

Command	Arguments	Returns	Comments
DTSeq			
create	<DTPkey> DTSkey		Create a DataTagSequence using a list of DataTagPair keys
createbyvalues	<Name Type value> DTSkey		Create a DataTagSequence using a list of contents of DataTagPairs
length	DTSkey	Length	Return the number of elements
names	DTSkey	<names>	Get all names of DataTagPairs in the DataTagSequence
geti	Index DTSkey DTPkey	Name [Prim./ Seq.] Type value	Return the contents of DataTagPair at index
getdtpairkey	Name DTSkey		Create a new DataTagPair using the element at the index
getvalue	Name DTSkey	Name [Prim./ Seq.] Type value	Return the contents of the DataTagPair from DataTagSequence with Name
getvaluekey	DTPkey Vkey		Same as above but store the Value against the key
add	DTSkey		Add a DataTagPair to the sequence
remove	[Index | Name] DTSkey		Remove the DataTagPair at index I or with name N and store it back
delete	DTSkey		

Table 2: DTSeq (DataTagSequence) command

1 and 2 list the commands used to access the DataTagPair and DataTagSequence as an example.

Example code to pass non-string variables in Tcl

```
DTPair create "Length" "long" 50
    testTag1
DTPair create "Width" "float"
    25.5 testTag2
DTSeq create "testTag1 testTag2"
    testDTSeq
set n [DTSeq length testDTSeq]
puts "Length of DTSeq (should be
    2): $n"
```

The first two lines create "DataTagPairs" with keys (variable names) 'testTag1' and 'testTag2'. Line 3 recalls from memory the 'testTag1' and 'testTag2' DataTagPairs and forms the DataTagSequence with the tag 'testDTSeq'. Lines 4 and 5 show some simple steps to do an operation on a data structure and display the results; in this case, get the length (number of elements) of the DataTagSequence and print it out.

Infrastructure for accessing different data structures

APC has a well defined hierarchy of well defined data types. To be able to handle all these different types of data structures, we have a global map (an object that keeps a list of names and associated data) for each of the different data types. When create operation is called for any particular data type, the created data type is stored in the map against the name/key passed in as an argument. The key is then used in any routine to pull the data from the map. The mechanism is local to each script executor and there is no naming conflict across scripts in the same plan. Also, each data structure has its own map and hence no naming conflict exists across different data types. Finally, to support sharing data between scripts in the same APC Plan, a plan-level map is used, and each script can copy data to and from that map.

3.3 CORBA method invocation commands

The second major category of new Tcl commands provides the script writer with the ability to invoke methods on other components of the APC Framework via CORBA. In a manner similar to the way data object commands were defined, one command per IDL interface was created. Each command uses the first argument as a switch to select what functionality of that component will be accessed. In general, one switch was defined for each logical interaction. These logical interactions were defined at the granularity that a script writer would need, and no finer. In some cases, there is a one-to-one correspondence between command + option and CORBA method, and in other cases many CORBA calls are wrapped together in one option.

Table 3 below shows an example command – this command handles interactions with the DataStore component. This is a good example where one option, for instance '`store`', results in multiple CORBA calls. In the case of '`store`', the script uses the 'store' option to put data into a database in a specific way. The C++ implementation of that command first invokes a 'find' command on the Data Store component to find if there is any similar data already stored in the database. If there is data already there, then the command uses a second CORBA method to replace the existing data with the new data. On the other hand, if nothing appropriate is in the database, the command uses an alternative CORBA method on the Data Store to create a new data set in the database.

By combining CORBA method invocations into a logical function, the extended scripting language can be kept relatively simple. The script writer doesn't deal directly with CORBA methods, just functionality that he/she needs to use.

3.4 Miscellaneous commands

In addition to commands to manipulate data objects and invoke methods on distributed objects, other utility commands were added. One need was for synchronization and communication between the main and event scripts. For synchronization the PE uses simple mutex locks – one script can set a lock and wait for another script (running in another process thread) to release the lock. In addition, the PE controls a global memory area that is separate from the memory used by each of the scripts. In order to exchange data, the scripts use this global memory through a new command which was added to allow the scripts to copy data to or from this global memory.

The example below illustrates two scripts using locks to synchronize their activities.

Example scripts

<main script>

```
    Lock create AlarmEventLock
    Lock wait AlarmEventLock 180
    if([Lock status]) {
    # event received
    ...
    } else {
    # timed out
    ...
    }
```

<event script>

```
    # do some processing
    Lock unlock AlarmEventLock
```

In this example, the main script needs to wait until the event script has run past a certain point before it continues executing. To accomplish this, the main script creates a lock called 'AlarmEventLock' and then waits for it to be released. When the event script reaches the `Lock unlock` command, it releases the 'AlarmEventLock', which allows the main script to continue. In case the event script never releases the lock for whatever reason, the main script times out after 180 seconds and continues with the rest of the

Command/Option	Arguments	Returns	Description
`DataStore`			
`store`	`[temp | perm] DTSKey NVSKey`		stores data in DataStore
`retrieve`	`[temp | perm] DTSKey [N | NVSKey]`	`{[Prim./ Seq.] Type value}`	retrieves only the exact matching data from the DataStore
`query`	`[temp | perm] DTSK`	`"contents of stored data"`	returns all partially-matching data stored in the DataStore
`delete`	`[temp | perm] DTSK`		deletes data stored in the DataStore

Table 3: Example CORBA method invocation command from the APC Framework

script.

In some cases, there are time-consuming activities that the scripts need to perform. These time-consuming commands typically contain CORBA calls, but in general can be any activity. In order to provide potential performance improvements, these time-consuming activities can be performed in the "background" for the cases where the command needs to be completed before some point is reached, but not necessarily in a deterministic order. A new command was written that allows a script writer to execute a Tcl command in a separate Tcl interpreter. This new interpreter is created and executed in a separate process thread from the calling script. This gives the calling script the ability to continue executing, while the separate interpreter handles the slower activities. Only a single command is allowed to be executed in this manner, but that command can be a procedure call, so just about any activities can be performed. This mechanism is ideal for slow setup processes and database access.

Example code

<sub-script>

```
proc setup_plugin {name} {
    set ex_plugin [PlugIn Setup
    $name]
    Move global ex_plugin
    ex_plugin_global
}
```

<main script>

```
set fork_wait [Fork setup_plugin
    "example_plugin"]
# do other things
...
# now wait for the background
task to finish
#(if it hasn't already)
Lock wait $fork_wait
```

The command `Fork` runs "*`setup_plugin "example_plugin"`*" in the separate Tcl shell. Prior to executing the command in the separate Tcl interpreter, the subscript file is evaluated in order to define any needed subroutines. The command also returns the name of the lock which will be released once the command has completed execution.

4. SUMMARY AND CONCLUSIONS

Tcl has proven to be a great base language upon which to build a CORBA scripting language. While the fact that in general Tcl deals only in strings may seem to hamper its use in such a distributed environment, the addition of specific commands that deal with the data types of interest keeps the command syntax and scripting simple and easy to learn.

The biggest drawback of embedding Tcl in such a multi-threaded component as the PE is its lack of thread-safety, which has begun to be addressed, eliminating the need for extensive modification of the Tcl core or locks around the library calls.

While this project started with Tcl 7.6, we have kept up to date with the recent Tcl releases. However, we have not been able to rewrite the extensions to utilize the object interfaces added in Tcl 8.0, so much improvements could be made to the APC extensions.

ACKNOWLEDGEMENTS

Portions of this work was performed under the support of the US Department of Commerce, National Institute of Standards and Technology.

REFERENCES

1. Orfali, R. , Harkey, D., Edwards, J., The Essential Distributed Objects Survival Guide, (Wiley, 1996).

DataMynah : A pseudo-NL interface to large multivariate datasets

De Clarke
University of California, Santa Cruz
de@ucolick.org, http://www.ucolick.org/~de

In a paper presented at Tcl97 [Clarke] I discussed a "knowledge based UI builder" called Dashboard. In this poster I present another in the family of applications built on a common database: DataMynah.

DataMynah uses the "data dictionary" or "Memes" portion of the knowledge base to initialize its internal context. After this initialization, the user can interact with DM using English-like commands to retrieve data from tables whose definitions are known to the knowledge base.

The advantage of the Memes information infrastructure over a generic (data type and column name) database UI is that Mynah can define and explain and show relationships between items. The user can "feel" his/her way into the data by question and answer rather than having to know the schema and nomenclature precisely in advance.

Mynah is not a true NL tool. It contains no generative grammar rules or elaborate parsers. The parser uses a brute-force substitution pass (similar to sendmail rules) followed by an "inspired guesswork" pass. In other words, it "understands" English sentences like a very naive non-native speaker, by grasping the meaning of some key words and phrases and trying to intepret the rest of the input in light of those known words.

Aside from the pseudo-NL interface, Mynah offers a hyper-text-like results/session-log window in which text can be interactive. It is peculiarly optimized for time series data, because the target real-world application is the analysis of large amounts of telemetry by hardware and software engineers. Hundreds of telemetry points are logged every N minutes or seconds; engineers will review these during post mortems or other inquiries into instrument and telescope performance. They will want to determine whether there is correlation between various states of the system and its failure modes. DataMynah easily allows extraction of data in standard formats for input into other tools (and can launch xgobi [XGobi] directly).

After a shallow investigation of tools like Bayesian correlation engines [AutoClass-C], I decided that for simple time-series information the human visual cortex was the optimal processing hardware, and that stacked stripcharts were the optimal representation. BLT was used to make the "stack-o-strips" windows which show selected telemetry points to the user on a consistent time axis (see Figure 1). Each log table may contain only part of the telemetry, but plots can be cut/pasted from one stack into another, to compare data from different sets.

This product is still in beta. Inital user reactions at a pre-release demo were guardedly positive. User requests were mostly for customization and shortcut features, and for more flexibility in the parser (improve the Turing illusion level).

The end product, if successful, will combine a documentary (answers questions about meaning and function of keywords and data tables) and analysis (access to rich telemetry) function. It could be applied to databases other than our own, and foreign language support would not be difficult due to the simplicity (stupidity?) of the parsing method. The combined NL/GUI approach is, I think, a more appropriate approach to complicated RDBMS UI than a strictly graphical one, which tends to become cluttered and overcomplex as the datasets increase in size and depth.

DataMynah is a pure Tcl/Tk application, though it can launch other visualization applications written in other languages. It was written using Tcl, Tk, TclX [TclX], BLT [BLT], TkTable [TkTable] and SybTcl [SybTcl]

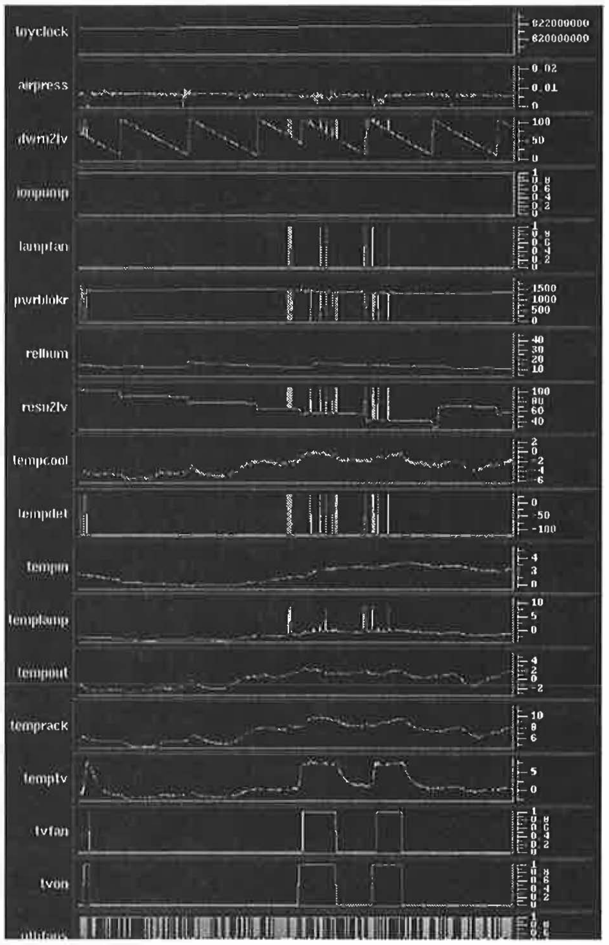


Figure 1: Time Series Data Displayed by DataMynah

The data in Figure 1 are telemetry data from 1995, ingested into the online RDBMS during alpha test of DM. In this stack of time-series plots, the coincident spikes in several of the telemetry sources jump right out to the human eye. It looks very much like noise bursts, and at first we thought we were seeing electrical switching noise. As it turns out, all the signals displaying the coincident spikes originate from the same ADC. The ADC is defective. This problem was not previously noticed because, despite the wealth of telemetry available, our methods of reducing and analyzing it were too laborious to make it worthwhile except for postmortems of catastrophic failures.

The purpose of DataMynah is to make monitoring easy and simple, so that failures of this kind are detected early. The series of user commands which resulted in this plot were approximately "I'm interested in temperature data for HIRES", "when I say ht I mean hitmplog,", "lookup ht", "get the ht from Dec 18 through Dec 26 1995", "chart d1". DataMynah has many other features which cannot be listed here; sample sessions and output will be available at the poster.

References

[Clarke] De Clarke, *'Dashboard': a knowledge-based real-time control panel*, Proceedings of the 5th Annual Tcl/Tk Conference, July 1998, pp 9-18

[AutoClass-C] AutoClass-C by Dr. Dianne Cook and Joseph Potts, **http://www.ph.tn.tudelft.nl/PRInfo/PRInfo/software/msg00089.html**

[BLT] BLT Tk extension by George Howlett, **ftp://ftp.tcltk.com/pub/blt/**

[SybTcl] SybTcl extension by Tom Poindexter, **http://www.neosoft.com/tcl/ftparchive/sorted/databases**

[TclX] Extended Tcl by Mark Diekhans and Karl Lehenbauer, **http://www.neosoft.com/tcl/ftparchive/TclX/**

[TkTable] TkTable extension by Jeffrey Hobbs, **ftp://ftp.cs.uoregon.edu/pub/tcl/tkTable/**

[XGobi] XGobi data visualization tool by Deborah Swayne, Dianne Cook, and Andreas Buja, **http://lib.stat.cmu.edu/general/XGobi/**

Enhancing Interfaces with Animated Widgets

D. Scott McCrickard
Graphics, Visualization, and Usability Center
and College of Computing
Atlanta GA 30332
mccricks@cc.gatech.edu, http://www.cc.gatech.edu/~mccricks/

As noted by Tcl creator John Ousterhout, scripting languages are designed to "glue" together existing resources making them easier and more efficient to use. In particular, the Tcl/Tk scripting language is designed to be platform-independent with an extensible graphical toolkit for interface design. However, many programs written with scripting languages (specifically *agents*) are autonomous and long-lived, implying that new interaction techniques may be necessary to best serve the interface designer. One approach is to use animation to provide a hands-off way to deliver information to a user.

Our Agentk toolkit consists of *animated widgets*, widgets whose appearance changes over time to provide additional information to users. An animated widget is a type of of *mega-widget*, a collection of widgets operated using a single programming interface. Programmers can control the contents, animation speed, bindings, and other factors of an animated widget using the same type of widget creation commands, widget subcommands, and options syntax that they use with existing widgets. Agentk consists of the following types of widgets:

- **Fade** widgets display multiple blocks of text or graphics within a given space by gradually fading between them. The information contained in a fade widget slowly changes color until it matches the background color. This fading effect allows multiple blocks of information (such as stock quotes or ball scores) to be displayed in a single area without having to flip or scroll between items. To control the rate at which information is displayed, a user can click and hold the mouse button to pause the fading effect, then can release it to jump to the next information block.
- **Ticker** widgets provide a ticker tape display that automatically scrolls or "tickers" text and graphics horizontally across the screen. As with the fade widget, a gradual tickering effect can be less distracting than a sudden switch in information. While the fade widget is useful for showing fixed-size blocks of information, the ticker widget proves to work better for streams of text with arbitrary length such as news or weather bulletins. The user can pause the flow of information by clicking on the ticker and can alter the flow by dragging.
- **Roll** widgets scroll information vertically across the screen. As one line disappears at the top of a widget the next appears at the bottom. Roll widgets are well-suited for ordered lists of information such as golf leaderboards or product rankings. As with the ticker widget, a user can pause or alter the flow of information using the mouse.
- **Navigation bar** widgets allow a user to view and find information in a list. Navigation bars employ many of the same components as a scrollbar (arrows, a thumb, a trough) that function in a similar manner, but the trough contains colored horizontal lines that represent the items in the list. The colors correspond to groupings of list items, so repeated color patterns will reveal related items. Navigation bars have various programmer-configurable layout options that are useful for various types of lists. As a list changes, the navigation bar automatically updates to show the change.

The Agentk toolkit has been used in a variety of applications for managing email, news, and Web-based information. More information, screenshots, and details on obtaining Agentk can be found at http://www.cc.gatech.edu/~mccricks/agentk/. We hope that the animated widgets in Agentk will help programmers build better information management agents and programs.

Mobile Streams

M.Ranganathan, Laurent Andrey, Virginie Schaal {mranga,andrey,schaal}@nist.gov
National Institute of Standards and Technology, Gaithersburg, MD 20899
Anurag Acharya
Dept. of Computer Science, University of California, Santa-Barbara, CA93106

Abstract

We present a toolkit for building event-driven, re-configurable distributed systems. An event-driven application is driven by asynchronous inputs that cause event-handlers to be invoked. A large class of distributed collaborative, testing, monitoring and control applications fit this paradigm - for example, conferencing and conference control applications, distributed control and testing applications and many others. In each case the notion of an "event" varies. In a collaborative system, events are user inputs; in a distributed monitoring and control system, events are changes in transducer inputs; in a distributed testing scenario, events are test outputs, timer alarms and so on. In such systems, it may be useful to have the ability to dynamically extend and re-configure the system. For example, in a collaborative system, different users may be interested in getting notification of different events that may not be known to the system designer a-priori – necessitating dynamic extension while the system is in execution. There are also situations where system design and performance may be enhanced by dynamic re-configuration - that is dynamic re-mapping of the system functionality while the system is in execution. Our goal is to build a system that enables the scripting of such event-driven applications.

Our basic abstractions consist of Mobile Streams (MStreams) and event handlers. A Mobile Stream is a named communication end-point in a distributed system that can be moved from machine to machine as computation is in progress while maintaining a well-defined ordering guarantee. The closest analogy to an MStream is an "active mobile mailbox". Like a mailbox, messages may be sent asynchronously to the MStream and are consumed in the same order in which they were sent. By attaching event handlers, message arrivals trigger Handler executions and hence the MStream becomes an "active mailbox". An MStream has a globally unique name and may be located on any machine that runs an execution environment for it and allows it to be moved there. The ability to move an MStream around makes it a "mobile mailbox". Handlers may be dynamically attached to (and detached from) an MStream and are independently and concurrently invoked for each event. Handlers operate in an atomic fashion. By "atomic" we mean that changes in the state of the MStream (which includes various attributes such as its location, the set of Handlers attached to it and so on) are deferred until the time when the Handlers complete execution. Handlers may append messages to other MStreams - triggering Handler executions at the target MStream when the message is delivered. Handlers are organized into groups (called Agents) with each group having its own interpreter and thread of execution and an MStream may have several Agents associated with it.

A distributed system is organized around its communication end-points i.e. MStreams, and by associating Agents with these end-points, which, in turn, attach Handlers for specific events. An Agent may specify a portion of its global state as being in its *briefcase* – indicating to the system that this state needs to be relocated to the new location when an MStream is moved.

Using the mechanism we have just described, an entire distributed system can be scripted and deployed from a single point of control and dynamically extended and re-configured while it is in execution. To set up such a system, the controller simply defines MStreams, associates handlers with the MStreams and moves the MStreams to the desired locations. This is useful in various scenarios such as web-based testing of distributed systems and conferencing where the web-server can set up the distributed test script for the entire test from a single point of control.

In scenarios involving multi-party collaboration, it is necessary to be able to place controls on how the system may be extended and re-configured. To allow this we have incorporated a resource-control mechanism. Each MStream can be created with its own resource-controller that allows the user to place restrictions on MStream opens, close, movement and so on. The system relies on daemons started at each participating site

to host an execution environment for MStreams. Each of these may be started with a script that permits controls to be placed on the resource usage at that site. In addition, we have a System Resource Controller for the entire system that allows the system designer to place controls on MStream creations, deletions and new peer additions.

Our prototype system called AGNI (Agents at NIST) is a Tcl-based multi-threaded system based on unmodified Tcl 8.1 (a2). Each group of handlers runs its own interpreter and has its own thread. Message delivery to Streams is via a custom reliable peer-to-peer message protocol built on top of UDP that preserves FIFO delivery order despite dynamic re-configuration. AGNI is currently operational and has been used to implement three fairly substantial applications – a toolkit for collaboratively sharing unmodified Tk applications (adapted from the TK-Replay code developed by Charles Crowley [Cr95]), a debugger for MStream programs (based on the freely available Tcl-Debug debugger developed by Don Libes [Li93]) and a monitor for visualizing an MStreams-based distributed system. Using our Tk-sharing toolkit, we were able to share a Tk-Drawing application[T98]. We have also developed a closely related application to record the interactions that occur in a collaborative conference and replay these interactions later. We have applied this technology to record and replay the user-GUI interactions that occur during the use of the MITRE XCVW collaborative tool. We are currently in the process of building a web-based tester for peer-to-peer applications that uses Mobile Streams as a run-time system. Other applications we are considering include a collaborative system for viewing of images from a large image data repository in telepathology applications and support for mobile proxys in ubiquitous computing applications.

References

[Cr95] C. Crowley, "Tk-Replay: Record and Replay in Tk", *USENIX Third Annual Tcl/Tk Workshop*, 1995.

[L93] D. Libes, "A debugger for Tcl " *Tcl/Tk workshop, 1993.*

[T98]http://www.inftechnik.tu-ilmenau.de/~silvio/research/soft.htm
Tk-Draw web page.

Tcl/Tk Program Development Tools

Clif Flynt (*clif@cflynt.com*)

Flynt Consulting Services, 9300 Fleming Rd., Dexter, MI 48130, USA

One of the tricks to getting your job done efficiently is having the right tools for the job. There are plenty of tools available for developing Tcl applications.

This poster gives a quick description of several tools that are in use in the Tcl community. If the tool you need is not mentioned here, try checking the Neosoft search engine at `http://www.neosoft.com`, the Scriptics Resource Center at `http://www.scriptics.com/resource/`, the announcements in `comp.lang.tcl`, or the FAQs at `http://www.teraform.com/%7Elvirden/tcl-faq/`.

The posters describe:

Code formatters	
`frink`	Reformats code into a standard style, for easy comprehension.
Primary Site	`ftp://catless.ncl.ac.uk/pub/frink.tar.gz`
`tcl_cruncher`	Reformats code into a style that optimizes for interpreter efficiency. This program also does some syntax checking.
Primary Site	`ftp://hplyot.obspm.fr:/tcl/tcl_cruncher*`
Code checkers	
`tclCheck`	Checks for balanced brackets, braces and parentheses.
Primary Site	`ftp://catless.ncl.ac.uk/pub/tclCheck.tar.gz`
`tcl_lint`	Checks for syntax errors, unset or non-existent variables, incorrect procedure calls, and more.
Primary Site	`http://icemcfd.com/tcl/ice.html`
`tclparse`	Checks for syntax errors, missing dollar signs, and other errors.
Primary Site	`http://www.informatik.uni-stuttgart.de\` `/ipvr/swlab/sopra/tclsyntax/tclparseHomeEngl.html`
Debuggers	
`Don Libes's Debugger`	This is a text oriented package with support for setting breakpoints, examining data, etc.
Primary Site	`http://expect.nist.gov/`
`tuba`	This is a GUI based package with multiple windows for both Tcl and Tk.

Primary Site	`http://www.doitnow.com/~iliad/Tcl/tuba/`
`TdDebug`	This GUI based package can attach itself to an already running Tk application.
Primary Site	`http://www.neosoft.com`
`TclPro Debugger`	This is a full featured, GUI based package from Scriptics that can debug remote or embedded applications as well as those on the local host.
Primary Site	`http://www.scriptics.com`

GUI generators

`SpecTcl`	SpecTcl creates a GUI skeleton for a Tk program.
Primary Site	`http://sunscript.sun.com/products/spectcl.html`
`Spynergy`	
Primary Site	`http://www.eolas.com`

Tcl Compilers

`ICEM Tcl Compiler`	The ICEM Tcl Compiler translates Tcl code into C code to improve performance.
Primary Site	`http://icemcfd.com/tcl/ice.html`
`Jan Nijtmans's plus-patch`	This patch applies some minor bug fixes, extends the shared library support, and makes it possible to convert Tcl scripts into executables that can run when Tcl is not installed.
Primary Site	`http://home.wxs.nl/~nijtmans/`
`Scriptics TclPro Compiler`	Generates Tcl bytecode files that can be evaluated by Tcl 8.0.3 interpreters with the appropriate extension.
Primary Site	`http://www.scriptics.com`

Tcl Extension Generators

`swig`	Swig creates Tcl extensions by reading the function and data definitions from an include file.
Primary Site	`http://www.cs.utah.edu/~beazley/SWIG/swig.html`

Tcl Packagers

`Scriptics TclPro Wrapper`	Wraps an interpreter, application script and ancilliary files into a single executable.
Primary Site	`http://www.scriptics.com`

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

ISBN 1-880446-98-7